VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Softwarové plagiáty - detekce pomocí otisků

Software Plagiarism - Using Fingerprint Detection

2015 Dorota Sikorová

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Dorota Sikorová**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Softwarové plagiáty - detekce pomocí otisků
Software Plagiarism - Using Fingerprint Detection

Jazyk vypracování: čeština

Zásady pro vypracování:

Detekce softwarových plagiátů a klonů je předmětem zájmu mnoha subjektů jak z praxe tak z akademické sféry. Jejich snahou je návrh a vývoj efektivních a přesných metod vedoucích k ochraně autorských práv k software. Cílem práce je výběr a implementace detekční metody založené na otiscích (fingerprintech) testovaného software.
1. Seznamte se s problematikou plagiátorství obecně. Poté se zaměřte na softwarové plagiáty a klony.
2. Seznamte se s klasifikací softwarových plagiátů, s dosud publikovanými přístupy k detekci softwarových plagiátů. Rovněž zpracujte přehled existujících aplikací.
3. Blíže se zaměřte na techniky využívající otisky (fingerprinty) .
4. Navrhněte a naimplemetujte aplikaci pro detekci plagiovaného software.
5. Navržené experimenty nad vhodnou kolekcí dat realizujte a vyhodnoťte. Výsledky porovnejte s výsledky obdobných metod.

Seznam doporučené odborné literatury:

[1] Cesare S., Xiang Y.: Software Similarity and Classification, Springer 2012, ISBN 978-1-4471-2909-7
[2] Chilowicz M., Duris E., Roussel G.:Syntax tree ngerprinting: a foundation for source code similarity detection. http://igm.univ-mlv.fr/LIGM/internal_report/pdf/2009_03.pdf

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **RNDr. Eliška Ochodková, Ph.D.**

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 31.07.2015 .....

Chtěla bych poděkovat všem, jež mi byli oporou v době přípravy a sepisování této diplomové práce.

Abstrakt

Tato diplomová práce se zabývá aktuálním tématem detekce plagiátu ve zdrojovém jazyce. V teoretické části jsou představeny základní pojmy plagiátorství, způsoby prevence i detekce. Nejsou opomenuty ani současné nástroje a systémy pro detekci plagiátu. Praktická část je naopak zaměřena na návrh a implementaci nového detekčního nástroje používajícího common intermediate language, program dependency graph a otisky. Šířeji se zde pojednává o výhodách zvolených přístupů i algoritmů a následném srovnání s jinými nástroji. Nakonec je provedeno testování a vyhodnocení úspěšnosti a přesnosti metody.

Klíčová slova: plagiát, plagiátorství, softwarové plagiátorství, detekce, C#, otisky, common intermediate language, program dependency graph, MinMax, Min Hash

Abstract

This thesis deals with issue of source code plagiarism detection. The theoretical part presents the fundamental concepts of plagiarism, prevention and detection and describes the current tools and systems for detecting plagiarism. The practical part is focused on the design and implementation of new detection tool using Common Intermediate Language, program dependency graph and fingerprints. Details of the procedure are described and advantages and disadvantages are mentioned. Finally, the testing is performed and evaluation of the success and accuracy of the method

Keywords: plagiarism, source code plagiarism, plagiarism detection, C#, fingerprints, common intermediate language, program dependency graph, MinMax, Min Hash

Seznam použitých zkratek a symbolů

AST	–	Abstract Syntac Tree
CIL	–	Common Intermediate Language
PDG	–	Program Dependency Graph
TED	–	Tree Edit Distance
GST	–	Greedy String Tiling
IL	–	Indermediate Language
NP	–	nedeterministický polynomiální
UC	–	use case diagram

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Plagiátorství neboli vydávání cizí práce za vlastní, provází lidstvo už odnepaměti. Hojně se s ním můžeme setkat u děl psaných přirozeným jazykem, avšak stále častěji se vyskytuje také u zdrojových souborů. Zatímco plagiátorství v přirozeném jazyce je poměrně dobře zmapovaná oblast, problematika plagiátorství ve zdrojových souborech je v porovnání s ostatními odvětvími velice mladá. Většina dnešních přístupů a nástrojů pro odhalování plagiátu softwaru se potýká buď s výkonnostními problémy, nebo neposkytuje dostatečně přesné výsledky. Mnoho z těchto metod se navíc soustřeďuje na porovnání pouze mírně modifikovaných plagiátu a v komplexnějších případech již nedokážou zdrojové kódy dostatečně porovnat. Cílem této práce je proto zmapování současně používaných plagiátorských technik, dostupných nástrojů pro odhalování plagiátu a v neposlední řadě návrh a implementace nového detekčního nástroje.

První kapitola této práce je zaměřena na osvětlení základních pojmů z oblasti plagiátorství. Mimo nezbytné definice jsou zde nastíněny také faktory pro rozpoznání plagiátu a nejpoužívanější detekční nástroje. V druhé kapitole je vymezen pojem softwarové plagiátorství. Pozornost je věnována zejména klasifikaci modifikací zdrojového kódu, detekčním metodám a algoritmům. Třetí kapitola má za cíl seznámit čtenáře s problematikou Common Intermediate Language. Pojednává zejména o jeho struktuře a možnosti využití v nástrojích pro detekci plagiátu. Čtvrtá kapitola se již věnuje popisu vlastního algoritmu, použitých metod a uvádí výhody a nevýhody zvoleného přístupu. Samotnému návrhu aplikace je věnována kapitola pátá. Nachází se zde analýza požadavků, případy užití i návrhové diagramy. Šestá kapitola plynule navazuje na předešlou a seznamuje čtenáře s implementačními podrobnostmi a problémy, jež při vývoji nastaly. Nakonec v kapitole sedmé lze nalézt výsledky testování a srovnání nově vytvořené metody s již existujícími nástroji.

2 Motivace

Plagiátorství je ve své podstatě forma krádeže duševního vlastnictví a tak jako každá jiná krádež by ani ona neměla být přehlížena. Vývoji jednoho softwarového produktu se často věnuje řada lidí, jež do něj vloží mnoho úsilí, snahy a vlastní invence. Neoprávněné kopírování nebo zneužití jejich práce představuje nejen porušení autorských práv ale také morálních zásad.

S rostoucím objemem volně přístupných dat se tvorba plagiátu stává stále jednodušší. Zdroje informací jsou dostupnější a techniky plagiátorů vynalézavější. Existuje řada nástrojů, jež dokáže odhalit napodobeniny vzniklé jednoduchou úpravou zdrojového kódu. Naneštěstí detekčních systému, jenž by odhalily složitěji maskované plagiáty je znatelně méně. Nejúčinnější z nich jsou navíc kvůli vyšší výpočetní náročnosti pro většinu uživatelů téměř nepoužitelné.

Cílem této práce je proto přispět ke zlepšení výše popsané problematiky tvorbou nového detekčního nástroje, jenž by byl schopný rozeznat i komplikovanější úpravy zdrojových kód s menší výpočetní náročností.

3 Taxonomie plagiátorství

Tato kapitola zodpoví několik základních otázek z oblasti plagiátorství a osvětli pojmy, jejichž definici je nutné znát pro pochopení dalších kapitol této práce.

Plagiátorství má mnoho podob a může se objevit v jakémkoli oboru, jenž vyžaduje kreativní myšlení. V rámci této diplomové práce se však zaměřím výlučně na plagiáty textu v přirozeném jazyce, o nichž pojednává tato kapitola a plagiáty zdrojového kódu, o nichž pojednává další kapitola.

3.1 Plagiátorství

Pod pojmem plagiátorství si lze představit doslovné nebo i částečné kopírování, reprodukci nebo znovupoužití cizího díla bez uvedení zdroje. V hojné míře se s ním setkáme například na akademické půdě, kde studenti ve svých pracích používají zdroje bez řádného uvedení v referencích. Mnoho studentů se také dopouští plagiátorství nevědomě chybným citováním zdroje nebo necitováním vlastních děl.

Plagiátor se svým počínáním dopouští hned několika přestupků. V první řadě porušuje morální standardy a také právní předpisy České Republiky. Neuvedením zdroje dále ochuzuje čtenáře díla o kontext, v jakém byl převzatý text uveden. Vytvořením plagiátu sám poukazuje na neschopnost, neznalost nebo nezájem o dané téma. Navíc plagiátorství může v krajních situacích vést až padělání původních dat a publikovaní dat mylných, zkreslených či dokonce v nejhorších případech vymyšlených.

3.2 Co je to plagiát

Plagiát představuje cizí dílo nebo jeho část, jež někdo jiný neprávem udává za své. Mezinárodní norma ČSN ISO 5127-2003 definuje plagiát jako „představení duševního díla jiného autora, půjčeného nebo napodobeného vcelku nebo zčásti, jako svého vlastního“[1].

Dílo lze dle [2] považovat za plagiát pokud obsahuje jednu nebo více chyb uvedených níže.

- úmyslný plagiát - za úmyslný plagiát lze považovat dílo, grafický prvek nebo text, jenž vznikl opsáním nebo okopírováním bez uvedení citace. Úmyslným plagiátem se také stává dílo, jež obsahuje okopírovanou strukturu cizího textu nebo naopak záměrně neobsahuje jeden nebo více zdrojů.
- nedodržení citační etiky - citovaný zdroj není uveden dostatečně a správně nebo není uveden zcela.
- chybná parafráze - dílo obsahuje převzaté myšlenky, aniž by byl citován jejich zdroj. Text díla byl vytvořen přeskupením výrazů původního textu, opět bez uvedení korektní citace.
- chybná kompilace - jedná se o použití pouze jednoho hlavního zdroje, vytvoření textu z doslovných pasáží různých zdrojů a necitování všech zdrojů

- necitování všeobecně známých faktů - autor necituje části textu, jež jsou dle něho všeobecně známy.
- necitování vlastních děl - častou chybou je také necitování vlastních děl. Mimo jiné tím čtenáře ochuzujeme o další zajímavý zdroj informací.
- kryptomnézie - autor použije myšlenku, jejíž původ si již nepamatuje

Naopak za plagiát se dle [2] nepovažuje:

- Dílo, jež vzniklo kompilací několika myšlenek a závěrů získaných z různých zdrojů. Nejedná se o doslovné kopírování celých úryvků. Kompilace podává ucelený pohled na danou problematiku a neobsahuje žádný nový poznatek. Použité zdroje i v tomto případě je nutné řádně a korektně citovat. Výsledná práce musí být prezentována jako kompilace a nesmí se nikdy vydávat za originál.
- Za plagiát není považovaná ani parafráze původního díla. V tomto případě dochází k přebírání pouze základních myšlenek.

3.3 Prevence plagiátorství

Prevenci plagiátorství lze provádět jak na straně autora původního díla, tak na straně autora nově vznikajícího díla.

Autor původního díla by neměl v rámci prevence nabízet své dílo k volnému použití. Pokud to přesto učiní je pravděpodobné, že jeho dílo bude použito jako předloha k nejednomu plagiátu.

Každý autor nově vznikajícího díla by se měl seznámit s definicí a základními principy plagiátorství. Díky těmto znalostem se vyhne mnohým nepříjemnostem a výsledné dílo nebude označeno za plagiát. Obecně lze říci, že je důležité, aby nová práce obsahovala vlastní prvky. Pokud je nevyhnutelně nutné použít část textu z jiného zdroje, je potřeba tuto přímou citaci ohraničit uvozovkami a uvést referenci na dílo, z něhož pochází. Referenci musíme použít také v momentě kdy, parafrázujeme text vlastními slovy nebo publikujeme překlad z jiného jazyka.

3.4 Detekce plagiátu v přirozeném jazyce

Přístup k elektronickým dokumentům je v dnešní době velice jednoduchý a zdrojů, z nichž mohou plagiáty vznikat, je nespočet. Vzhledem k těmto faktům a důvtipnosti parafrázujících autorů je odhalování plagiátu v přirozeném jazyce velice náročný úkol.

Dílo lze považovat za podobné, pokud obsahuje jeden nebo více níže uvedených faktorů

- shodné nebo podobné bloky textu
- podobnou strukturu dokumentu

- náhlé změny stylu psaní – například změny konstrukce vět
- gramatické chyby ve shodných slovech
- podobná často se opakující slova

Manuální porovnání přichází v úvahu pouze u malého množství dokumentu, u něhož autor nemůže čerpat z mnoha již dostupných informací. U většího množství dat ztěžuje porovnání dříve zmiňované parafrázování a také použití několika cizích zdrojů informací. U větších objemu dat tedy přichází na řadu automatické systémy, které není tak lehké oklamat. Jsou schopny odhalit plagiáty vyhledáním podobných bloků textů, struktury dokumentu, gramatických chyb a často se opakujících slov. Avšak ani na tyto detekční nástroje nelze zcela spoléhat, jelikož se mohou dopouštět chyb.

V dnešní době máme k dispozici hned několik systému pro detekci plagiátu v přirozeném jazyce. Pro porovnání prací v jazyce českém jsou nejpoužívanějšími programy theses.cz (dostupný z `www.theses.cz`) a odevzdej.cz (dostupný z `www.odevzdej.cz`). Za jedny z nejlepších zahraničních detektorů plagiátu jsou považovány ithenticate.com (dostupný z `www.ithenticate.com`) a turnitin.com (dostupný z `www.turnitin.com`).

4 Softwarové plagiátorství

Na rozdíl od předešlé kapitoly, která pojednávala o plagiátorských technikách spíše v obecné rovině, se v této části budeme zabývat výhradně plagiáty softwaru. Nejprve bude uvedena definice plagiátu softwaru a následně budou představeny nejpoužívanější metody a algoritmy používané pro jeho odhalování.

4.1 Definice plagiátu zdrojových kódu

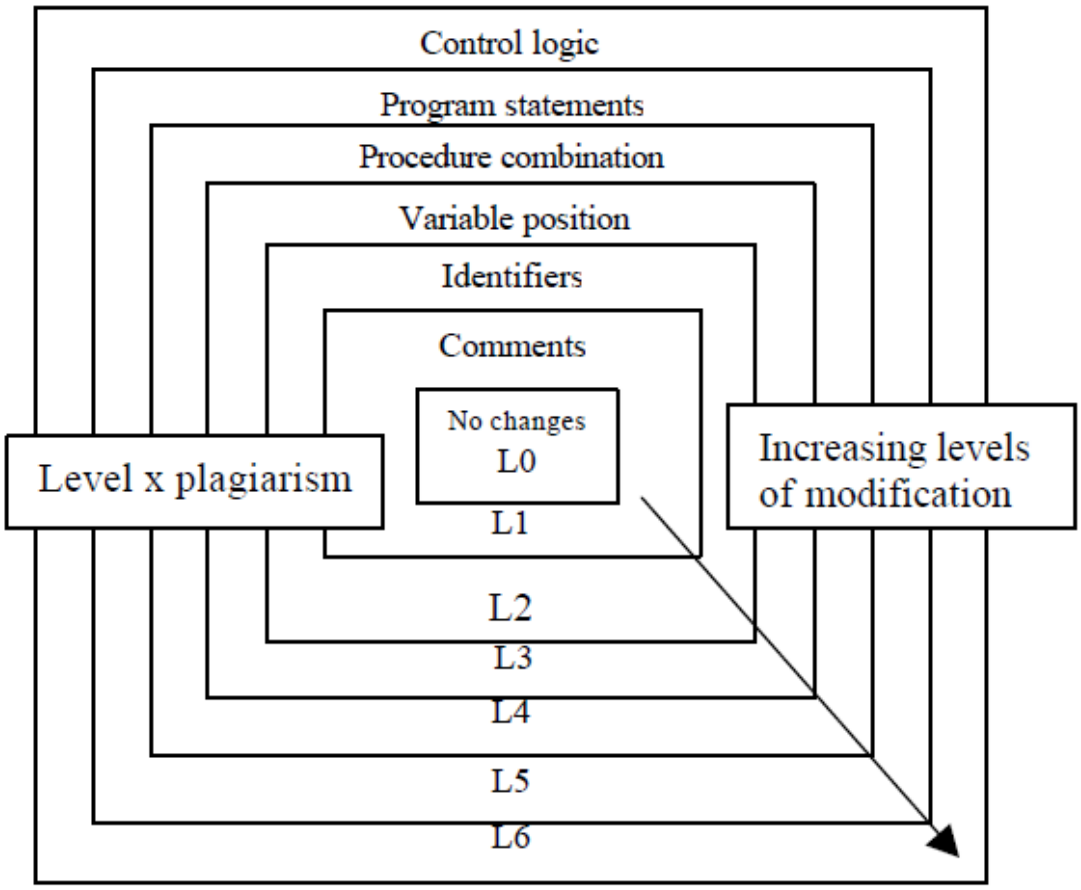
Definice plagiátu zdrojového kódu se příliš neliší od plagiátu textů uvedené v kapitole 3.2. Autoři díla Computer Algorithms for Plagiarism Detection (1989) Alan Parker a James Hamblen definují plagiát zdrojového kódu jako „program, jenž byl vytvořen na základě jiného zdrojového kódu s malým počtem změn“. Jedná se tedy v podstatě o přebírání nebo dokonce kopírování zdrojových kódů bez výslovného souhlasu majitele a vydávání těchto kódu za své vlastní.

Může se však jednat také o použití kódu bez reference na původního majitele. Tato situace často nastává při použití takzvaných „open source“ programů jejichž licence sice dovoluje použití souborů bez poplatku, avšak požaduje zachování jmen autorů a prohlášení o autorských právech přímo v kódu.

4.2 Modifikace zdrojových kódu

Cílem každého plagiátora softwaru je upravit zdrojový kód tak, aby byl plagiát zcela neodhalitelný a plagiátor jej mohl beze strachu vydávat za svůj. Přesto má mnoho plagiátu několik společných rysů.

Dle Clougha [3] lze modifikace měnící strukturu a podobu kódu definovat v několika kategorizovaných úrovních od nejjednodušších úprav, mezi které patří například změny názvů až po ty komplexnější zahrnující úpravy kontrolní struktury programu.



Obrázek 1: Úrovně modifikace plagiátu dle Clougha [3]

Úroveň L0 představuje situaci, kdy plagiátor zkopíruje kód a neprovede na něm žádné další úpravy. Jedná se bezpochyby o nejjednodušší způsob nevyžadující žádné programovací znalosti, avšak je také nejsnáze odhalitelný.

Úroveň L1 zahrnuje základní úpravy formátování kódu. Nejčastěji zde dochází ke změně nebo případně smazání prázdných řádků, komentářů, mezer a tabulátoru.

Do úrovně L2 lze zařadit změny názvu proměnných a metod. Ani tato technika není příliš efektivní a plagiát je, stejně jako v předešlých dvou případech, lehce odhalitelný.

Úpravy deklarace proměnných zahrnuje úroveň L3. Může se jednat o změnu datového typu proměnné, změnu umístění deklarací proměnných nebo například změnu pořadí operandů v příkazu aniž by došlo k ovlivnění výsledku.

V rámci úrovně L4 dochází již ke změně pořadí metod nebo funkcí. Často používanou praktikou je nahrazení volání metody jejím tělem a naopak nebo sloučení více metod. Tento způsob dokáže změnit vizuální stránku kódu a zmást nejeden detekční systém.

Úroveň L5 je mnohem sofistikovanější, jelikož zahrnuje významnější zásahy do původního zdrojového kódu a ovlivňuje nejen jeho vizuální ale také syntaktickou podobu. Oblíbenou technikou je vkládání nadbytečných metod a příkazů, které pouze vytvářejí iluzi obsáhlejšího kódu, avšak na funkčnost nemají žádný vliv. Častou úpravou je také přepisování struktur a vkládání nedosažitelného kódu. Tato úroveň vyžaduje, aby plagiátor měl alespoň základní znalosti programování. Pro odhalení plagiátu je zde již nutné použít jeden z detekčních programů.

Nejnáročnější je z hlediska úprav úroveň L6, která mění syntaktickou strukturu části nebo celého programu.

Pro doplnění je vhodné uvést také klasifikaci modifikací zdrojových souborů dle Jonese [7], který za plagiát považuje dílo, jež obsahuje:

- doslovné kopírování
- změny komentářů
- změny formátování
- přejmenování identifikátorů
- změny v uspořádání kódu a pořadí operandů
- změny datového typu
- redundantní informace
- záměna syntaktických struktur

Ukázku plagiátu zdrojového kódu lze vidět na obrázku 2. Jedná se o jednoduché změny, jenž nemají na běh algoritmu žádný vliv ale pro plagiáty jsou přitom charakteristické.

```
//Pr1 - Výpočet Fibonacciho čísla pomocí iterace
//Pr2 - Originální název metody
static int FibonacciIterative(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;

    int prevPrev = 0;
    int prev = 1;
    int result = 0;

    for (int i = 2; i <= n; i++)
    {
        result = prev + prevPrev;
        prevPrev = prev;
        prev = result;
    }
    return result;
}

static void PrintOutputText(int n)
{
    if(n<100)
    {
        Console.WriteLine("Výsledek "+n+" je menší než 100.");
    }else
    {
        Console.WriteLine("Výsledek " + n + " je větší než 100.");
    }
}

static void Main(string[] args)
{
    //Pr3 - Deklarace lokální proměnné
    int n = 10;

    var result = FibonacciIterative(n);

    //Pr4 - Volání metody
    PrintOutputText(result);
}
```

```
//Pr3 - Deklarace globální proměnné
public static int number = 10;

//Pr1 - Výpočet Fibonacciho čísla pomocí rekurze
//Pr2 - Přejmenování metody
static int FibonacciRecursive(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;

    return FibonacciRecursive(n - 1) + FibonacciRecursive(n - 2);
}

static void Main(string[] args)
{
    var result = FibonacciRecursive(number);

    //Pr4 - Tělo metody
    if (result < 100)
    {
        Console.WriteLine("Výsledek " + result + " je menší než 100.");
    }
    else
    {
        Console.WriteLine("Výsledek " + result + " je větší než 100.");
    }
}
```

Obrázek 2: Ukázka plagiátu zdrojového kódu

Plagiát zobrazený na pravé straně obsahuje oproti originálu vlevo několik úprav. Nejnáročnější a zároveň nejhůře odhalitelnou modifikací je bezpochyby změna iterace za rekurzi. Dále je v kódu patrná záměna těla kódu za volání metody. Méně závažnou úpravou je pak přidání komentářů a přepsání lokální proměnné na globální.

4.3 Detekční metody

Metody pro detekci plagiátu softwaru lze zařadit do dvou kategorii – ty, jež jsou založené na porovnávání počtu atributů a metody, které porovnávají strukturu kódu. Několik následujících podkapitol se podrobněji věnuje jednotlivým představitelům těchto detekčních přístupů. Uváděny jsou jak jejich pozitiva, tak i nedostatky.

4.3.1 Počítání atributů

Jedny z prvních nástrojů pro detekci plagiátu využívaly právě metodu počítání atributů. Míra podobnosti je v tomto případě určena na základě počtu výskytů klíčových slov, cyklů, řídících příkazů, řádků kódu nebo proměnných.

Typickým představitelem metody počítání atributů je Halstead [8]. Halstead ve své práci definoval metriky pro zjištění míry podobnosti (kapitola 4.4.1), avšak již se nezabývá samotnými operátory a operandy, jejichž popis je „intuitivně zřejmý a nevyžaduje

další vysvětlení“ [8]. V praxi, pro účely měření podobnosti, je intuice nedostatečná a neposkytuje přesné a především reprodukovatelné výsledky. Přitom žádný ze současných přístupů není obecně vnímán jako nejefektivnější a smysluplný způsob jak tyto parametry klasifikovat a následně počítat [9]. Případní zájemci o použití této metody jsou odkázaní na vlastní interpretaci nebo již existující strategie uvedené například v dílech "An Attribute Grammar Approach to specifying Halstead's Metrics"[10], "Defining Software Science Counting Strategies"[11] nebo Software Engineering Metrics and Models [12]. Podrobnější analýzou definice a návrhu Halsteadovy metriky se zabývá například Rafa E. AL QUTAISH a Alain ABRAN ve svém díle „An Analysis of the Design and Definitions of Halstead's Metrics „.

V současné době se již počítání atributů pro porovnání zdrojových kódu příliš nepoužívá. Hlavním důvodem je výše zmiňovaná nejednoznačnost pojmů, jenž může výsledky velice ovlivnit a také nepřesnost. Počet výskytů ať se již jedná o operátory, cykly nebo klíčová slova může být známkou plagiátu, ale také nemusí. Zvláště pokud plagiát vznikl rozšířením původního kódu o dodatečné a mnohdy i zbytečné řádky.

4.3.2 Porovnání struktury

Na rozdíl od počítání tokenů vykazují metody porovnávání struktur mnohem přesnější a relevantnější výsledky. Pro zjištění míry podobnosti jsou porovnávaný samotné struktury kódu. Tento proces je obvykle prováděn ve dvou fázích:

- Předzpracování – zde dochází k odstranění bílých znaků, komentářů, atd. a následně k vygenerování posloupnosti tokenů, které popisují příkazy nebo případné bloky příkazů.
- Porovnání – zde jsou tokeny převedeny do požadované struktury (AST, PDG, apod.) a je vypočtena podobnost na základě zvoleného algoritmu pro porovnání.

4.3.2.1 Porovnání řetězců

Metoda porovnání řetězců určuje míru podobnosti na základě vyhledání společných posloupností podřetězců. V první fázi dochází k odstranění nadbytečných částí, převedení veškerých písmen na malá a nahrazení identifikátorů (např. názvy metod nebo proměnných) za stejný token. Na takto předzpracovaný kód je aplikován algoritmus pro vyhledání společných podřetězců. Pokud je v porovnávaných souborech nalezen určitý počet shodných částí, jsou programy vyhodnoceny jako podobné. Tento přístup odhalí nejčastější a nejjednodušší metody pro zamaskování plagiátu.

Existuje celá řada algoritmů, jež se zabývají právě touto metodou [13, 14]. Jako poměrně efektivní lze označit Berry-Ravindran algoritmus [16], úspěšné jsou také jeho modifikace TVSBS [15] a SSABS [17] algoritmy.

Doba porovnávání kódu je oproti jiným metodám poměrně krátká, avšak výsledky ani v tomto případě nejsou příliš spolehlivé. Navíc je lze znatelně znehodnotit vložením nového nebo odstraněním existujícího kódu, případně přejmenováním proměnných. Z

toho důvodu porovnání řetězců není použito jako hlavní metoda v skoro žádném systému pro detekci plagiátu. Využívá se však často jako jakýsi doplněk k efektivnějším algoritmům [14].

4.3.2.2 Porovnání tokenů

Tato metoda je založena na porovnání sekvence tokenů získaných ze zdrojového souboru procesem zvaným tokenizace. Zdrojový soubor je nejprve předzpracován za účelem odstranění nadbytečných informací, které nemají vliv na výsledek a následně se jednotlivé řádky zdrojového kódu dle lexikálního pravidla použitého programovacího jazyka rozdělí na tokeny. Příklad zdrojového kódu a jeho tokenizované formy lze vidět v tabulce 1. Z takto zpracovaných souborů a tříd se vytvoří sekvence tokenů. Podobnost je následně určena na základě zvoleného algoritmů např. Jaccardovy, Diceho nebo kosinové podobnosti.

Zobecnění kódu, ke kterému dochází při tokenizaci, napomáhá k odhalování plagiátorských technik jako přetypování datových typů proměnných. Metoda však není odolná oproti náročnějším změnám ve struktuře kódu.

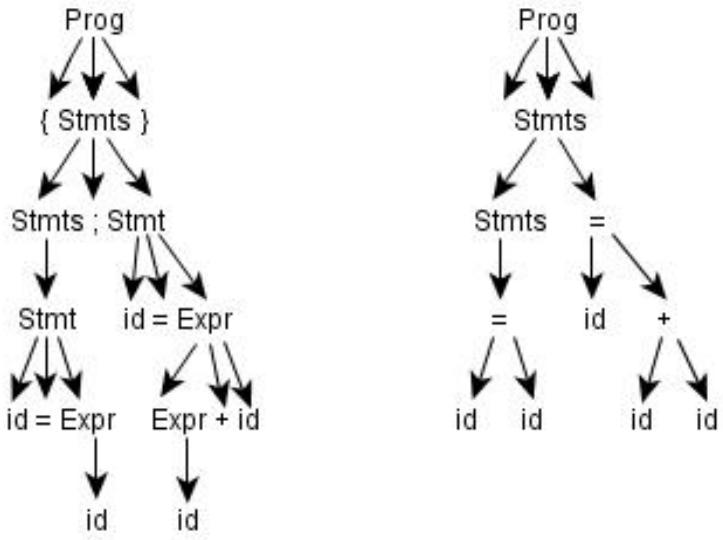
Zdrojový kód	**Tokeny**
string a = "";	Přiřazení
a = Console.ReadLine();	Vstup
if(a=="Y")	Podmínka, Porovnání
{	Začátek
SpustProgram();	VyvolejFunkci
Console.WriteLine("Program spuštěn.");	Výstup
}	Konec

Tabulka 1: Ukázka tokenizace

4.3.2.3 Abstraktní syntaktický strom

Abstraktní syntaktický strom (AST) představuje stromovou reprezentaci syntaktické struktury kódu, která obsahuje pouze ty prvky jazyka, jež ovlivňují sémantiku programu. Jednotlivé uzly AST tvoří konstrukce programovacího jazyka (např. operátory, volání, atd.) a jeho potomci reprezentují parametry. Konstanty, proměnné a literály jsou reprezentovány listy stromu. Na obrázku 3 lze vidět ukázku AST stromu.

Abstrakce AST stromu spočívá ve faktu, že neobsahuje veškeré detaily skutečné syntaxe programu. K zachycení skutečné syntaxe slouží konkrétní neboli parsovací strom. Jeho uzly představují neterminální pravidla bezkontextové gramatiky a listy zastupují syntaktické jednotky. Díky tomuto rozložení strom obsahuje všechny tokeny vytvořené lexikální analýzou. Parsovací strom bývá používán jako intermediální forma AST. Odstraněním redundantních informací, které nepřinášejí podstatné informace o struktuře programu, vznikne z parsovacího stromu AST.



Obrázek 3: Ukázka AST (v levo) a parsovacího stromu(v pravo)

Porovnání zdrojových kódu může v tomto případě probíhat jedním ze dvou způsobů. Prvním je porovnání uzlu s uzlem. Tento způsob však není příliš efektivní, zvláště pokud kontrolujeme větší množství dat. Porovnání mezi zdrojovými kódy může také probíhát na základě vyhledání podobných podstromů. Tento proces však bývá také poměrně časově náročný a u rozsáhlých programů je téměř nepoužitelný. Poslední možností je hybridní řešení, tedy spojení AST s jiným přístupem k odhalování plagiátu například vytváření otisků [18]. Výpočetní náročnost se díky tomu sníží a výsledky budou stále relevantní.

Důvodem proč je AST oblíben a použit v tolika algoritmech pro detekci plagiátu [19, 20] je bezpochyby odolnost nejen vůči základním plagiátorským technikám ale i používání alternativní syntaxe (přepsání cyklu for za foreach, přehazování case sekvencí, atd.). Podstatnou nevýhodou je však slabost proti vkládání nového nebo nadbytečného kódu.

4.3.2.4 Vytváření otisku

Otisk představuje jakýsi identifikátor dat, jež byly převedeny do mnohem menší formy. Identifikovat takto můžeme jakákoli data, avšak žádné dvojici různých informací by neměl náležet stejný otisk. Ke kolizím by mělo docházet pouze výjimečně u ojedinělých případů.

Mezi nejčastěji používané funkce vytvářející otisky patří hashovací funkce které generují takzvaný hash. Díky jeho vlastnostem a velikosti lze efektivně porovnávat i větší množství zdrojových kódu. Nežádoucí jevem je, že i při minimální změně vstupních dat

se zcela změní i otisk. Tato vlastnost ztěžuje nalezení plagiátu, jelikož podobnost a vzdálenost jsou pro porovnání dvou souboru nezbytné.

Další velmi oblíbenou funkcí je Winnowing jejíž hlavní myšlenka byla představena v díle „Winnowing: Local Algorithms for Document Fingerprinting“ [5]. Podrobněji se tímto algoritmem zabývá kapitola 4.4.4.

Vytváření otisku je hojně používáno jako dodatečná metoda pro zkrácení času výpočtů a zmenšení množství dat nutných k porovnání. Pro potřeby počítání atributů může nést informace o souboru a všech jeho atributech. Výborně se uplatní také u AST stromu, kde poslouží k zjednodušení porovnávání podstromů. Pro budoucí potřeby porovnání nemusí být ukládány celé stromy, ale pouze jejich otisky což představuje značnou úsporu místa. Stejné vylepšení poskytnou také metodám založeným na porovnávání různých typů grafů například grafů závislosti podgrafů (PDG).

4.3.2.5 Graf závislosti programu

PDG představuje grafické zobrazení funkcí, metod a procedur. Uzly zastupují příkazy zdrojového kódu a hrany představují tok dat, respektive použití dat v daném příkazu.

Oproti ostatním metodám PDG nezachycuje syntaxi, nýbrž tok dat. Vzhledem k tomu, že syntaktické změny (zvláště přepsání cyklu a přetypování proměnné) patří k nejčastějším operacím, vykazuje PDG, na něhož tyto změny nemají vliv, výborné výsledky.

Algoritmus vychází z myšlenky, že plagiátor sice může zdrojový kód libovolně upravit, avšak jeho správnost musí být zachována. Přestože by jí velmi kreativní plagiátor mohl obelstít, náročnost takových úprav by ve většině případu převyšovala náročnost vytvoření vlastního kódu.

PDG má však také své nevýhody. Při vyhledávání podobných podgrafů narážíme na isomorfismus podgrafů, který je považován za NP-úplný problém. Je potřeba tedy snížit počet podgrafů a aplikovat filtry pro jejich zjednodušení.

O úspěšnosti této metody hovoří mimo jiné její aplikace v jednom z komerčních systému GPlag [21]. Autoři tohoto díla zveřejnili v jednom ze svých článků [21] porovnání s ostatními plagiátorskými metodami.

	Porovnání řetězců	**AST**	**Porovnání tokenů**	**PDG**
Změny formátování	Ano	Ano	Ano	Ano
Přejmenování identifikátorů	Ne	Ano	Ano	Ano
Přehazování příkazů	Ne	Ne	Částečně	Ano
Změna řídících bloků	Ne	Ne	Částečně	Ano
Vkládání kódu	Ne	Ne	Ne	Ano

Tabulka 2: Porovnání odolnosti detekčních metod

4.4 Algoritmy používané pro detekci podobnosti

V zájmu získání relevantních dat za přijatelnou dobu většina ze současných detekčních systému kombinuje několik technik a algoritmů. V této kapitole jsou uvedeny nejznámější z nich spolu se stručným objasněním a výčtem jejich výhod i negativ.

4.4.1 Halsteadovy metriky

Všechny Halsteadovy metriky jsou založeny na principu rozložení programu na množiny operandů a operátorů. Podobnost programu pak lze spočítat například dle následující metriky:

$E = (N_1 + N_2) * \log(n_1 + n_2)$

Kde:

- N_1 – počet jedinečných operátorů
- N_2 – počet jedinečných operandů
- n_1 – počet operátorů
- n_2 – počet operandů
- $N = N1 + N2$ - představuje implementační délku
- $n = n1 + n2$ - slovník
- $V = N * log2(n)$ - za pomocí hodnot implementační délky a slovníku je dále vypočtena velikost souboru neboli „volume“

Halsteadova metrika bývala hojně používaná u prvních detekčních nástrojů, které využívaly metodu počítání atributů jako je tomu u 4.5.2.

4.4.2 Greedy String Tiling

Greedy String Tiling (GST)[26] byl navržen M. J. Wisem v roce 1993 a byl použit v detekčním systému JPlag, YAP i Plaggie. Algoritmus GST porovnává dva textové řetězce ve dvou fázích, které se opakují tak dlouho, dokud nejsou nalezeny žádné shody.

- V první fázi dochází k tokenizaci neboli převedení textu do sekvence tokenů. K převodu je využit lexikální slovník obsahující slova adekvátní k řešenému problému tedy v tomto případě se jedná o klíčová slova programovacího jazyka.
- Druhá fáze stanovuje míru podobnosti porovnávaných souboru.

Algoritmus stanovující míru podobnosti v rámci druhé fáze musí splňovat následující body:

- Každý token v řetězci se smí porovnat pouze jednou
- Transponované podřetězce by neměly mít žádný vliv na hodnotu podobnosti
- Rozkládání komplexnějších příkazů by nemělo ovlivnit výslednou hodnotu podobnosti

```
length_of_tokens_tiled := 0
   Repeat
      maxmatch := minimum_match_length

      /* Pro kazdy neoznaceny token podretezce P */
      for each Pp do
         /* Pro kazdy neoznaceny token podretezce T */
         for each Tt do
            j := 0

            while (Pp + j AND unmarked(Pp + j) AND unmarked(Tt + j)) do
               j := j + 1

            /* Pricte shodu do listu shod */
            if (j == maxmatch) then add match(p, t, j)

            /* Pokud je delka soucasne shody vetsi nez dosavadni */
            else if (j > maxmatch) then start new list with match(p, t, j) and maxmatch := j

      for each match(p, t, maxmatch) in list

         if not occluded then
            for j := 0 to maxmatch − 1 do
               mark_token(Pp + j)
               mark_token(Tt + j)
             length_of_tokens_tiled := length_of_tokens_tiled + maxmatch;

      Util maxmatch = minimum_match_length
```

Výpis 1: Pseudokód Greedy String Tiling algoritmu

Na výpisu 1 lze vidět Pseudokód GST pro porovnání dvou řetězců dle M. Wise. Nejprve jsou vyhledány všechny společné podřetězce díky porovnávání tokenů. Cyklus porovnává tokeny prvního a druhého souboru tak dlouho dokud se tokeny shodují. V momentě kdy se liší nebo je jeden z nich již označený zaznamená se délka aktuální posloupnosti. Pokud je tato délka rovna nejdelší zjištěné délce neboli takzvané maxmatch je tato posloupnost tokenů přidána do množiny matches. Pokud je však délka nově nalezené posloupnosti větší než maxmatch je množina matches znovu inicializována tak aby obsahovala pouze nově nalezenou posloupnost a do proměnné maxmatch je uložená její délka. Ve druhé fázi dochází k označování tokenů z množiny matches. Cyklus REPEAT se opakuje tak dlouho, dokud není aktuální délka nejdelšího shodného řetězce rovna minimální délce shody. Výstupem celého procesu je seznam nejdelších společných řetězců. Pro vypočítání míry podobnosti lze použít vzorec:

$$sim(P,T) = \frac{2*coverage(tiles)}{|S|+|T|}$$

$$coverage(tiles) = \sum_{match(p,t,length)\in tiles} length$$

GST algoritmus bohužel může dosahovat polynomiální složitosti O(n3), kde n představuje počet tokenů v řetězci. Pro zvýšení výkonu je tedy spíše doporučována jeho modifikace Karp-Rabinův GST s průměrnou složitostí O(n) nebo Running Karp-Rabin GST(RKR - GST).

4.4.3 Running Karp-Rabin Greedy String

Tento algoritmus vytvořený Richardem M. Karpem and Michaelem O. Rabinem v roce 1987 představuje kombinaci algoritmů Karb-Rabin a výše zmiňovaného Greedy String Tiling. Jelikož je GST algoritmu věnovaná předešlá kapitola, bude tato zaměřená na objasnění Karp-Rabinova algoritmu a jeho rozšíření.

Rabin–Karp algoritmus byl určen pro vyhledávání podřetězců. Na rozdíl od samotného GST však nepracuje s tokeny ale otisky ve formě hashe. Jeho průběh lze rozdělit do několika fází:

- V první fázi dochází z vyhledání všech podřetězců stejné nebo větší délky než je délka definovaná uživatelem. Každému podřetězci je následně vytvořen hash.
- V druhé fázi dochází k porovnání hodnot hashů. Pokud je nalezena shoda porovnají se řetězce ještě token po tokenu pro ověření, zda se opravdu jedná o shodu a nedošlo pouze ke kolizi hashů.

Running Karp-Rabin rozšiřuje Karb-rabinův algoritmus následujícím způsobem:

- V druhé fázi stále dochází k porovnání hodnot hashů. Pro zmenšení maximální složitosti O (n2) takovéhoto výpočtu se však zde používá hashovací tabulka. Díky ní je možné vyhledat umístění všech podřetězců, jež jsou shodné s ověřovaným řetězcem.

4.4.4 Winnowing

Winnowing patří mezi algoritmy vyhledávající podřetězce založené na vytváření otisků k-gramů. K-gram v tomto případě představuje souvislý podřetězec pevné délky k.

Winnowing funguje v několika fázích, které jsou zachyceny na ukázce 4.1. V podstatě rozdělí vstupní řetězec do k-gramů a každému z nich přidělí hash. Ty jsou dále spolu s informacemi o dokumentu použity pro vytvoření unikátního otisku dokumentu. Pro zvýšení efektivity algoritmu se v praxi často nevytváří otisk ze všech hashů ale pouze jejich podmnožiny. Ta je často volena na základě matematické funkce, nejčastěji 0 mod p, kde p představuje konstantu. Do otisku lze samozřejmě přidat další zpřesňující informace. Detekční nástroj MOSS například vychází z přesvědčení, že nejmenší hash by měl být u obou podobných souborů shodný a proto jeho hodnotu přidává do každého otisku[27].

Příklad 4.1
`A do run run run, a do run run`
Náhodně zvolen text.

`adorunrunrunadorunrun`
Předzpracovaný text bez bílých znaků

```
adoru dorun orunr runru unrun nrunr runru
unrun nruna runad unado nador adoru dorun
orunr runru unrun
```
Sekvence 5-gramů textu

```
77 72 42 17 98 50 17 98 8 88 67 39 77 72 41
17 98
```
Sekvence zahashovaných 5-gramů

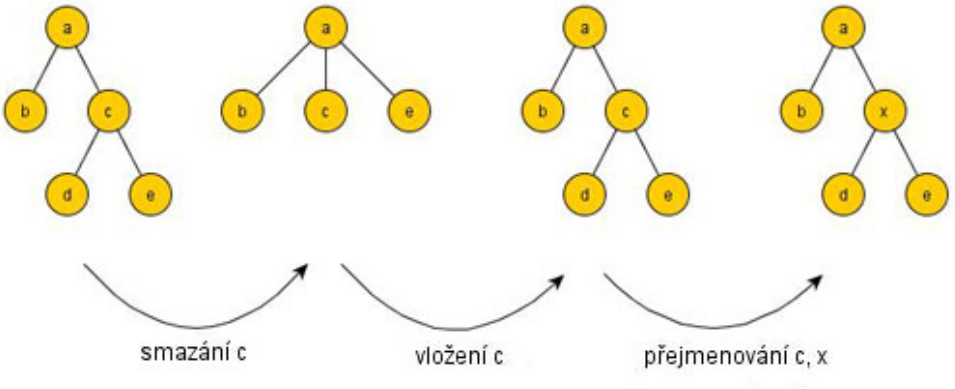
```
72 8 88 72
```
Sekvence hashů zvolených pomocí funkce $0 mod 4$.

■

Pokud dva dokumenty obsahují jeden nebo více shodných hashů je velmi pravděpodobné, že obsahují také stejné k-gramy. Konkrétní míru pravděpodobnosti souborů lze tedy stanovit poměrem shodných a rozdílných hashů.

4.4.5 Tree edit Distance

Tree edit distance (TED) neboli v překladu editační vzdálenost stromu je algoritmus pro zjištění míry podobnosti dvou stromových struktur. V podstatě udává minimální počet základních editačních úprav, jenž je nutné provést k tomu, aby se jeden strom transformoval na druhý. Mezi základní úpravy lze přitom řadit smazání, přidání nebo přejmenování vrcholu. Pokud je prováděno odstranění vrcholu, kterýnení listový musí nejprve dojít k odstranění všech uzlů jeho podstromu.



Obrázek 4: Ukázka transformace TED

Vzhledem ke složitosti tohoto algoritmu O(mn) není příliš vhodný pro větší objemy dat. Jako řešení se opět nabízí použít jednu z doplňkových metod například vytvoření otisků.

4.4.6 Min Hash

Min hash představuje efektivní způsob pro zjištění míry podobnosti mezi dvěma grafy za použití malého množství dat. Původně byla myšlenka tohoto algoritmu prezentována v práci Cohena [Cohen 1997]. Na porovnávaná data aplikuje hned několik hashovacích funkcí avšak pro porovnání bere v potaz pouze jejich minimální výsledné hodnoty. Výsledkem funkce je otisk tvořený vektorem číselných hodnot. Na určení miry podobnosti lze použít například algoritmus MinMax[28], který vznikl zobecněním Tanimoto funkce pro výpočet délky cesty v grafu. Udává poměr mezi počtem sdílených cest a celkovým počtem cest, který lze vyjádřit také vzorcem:

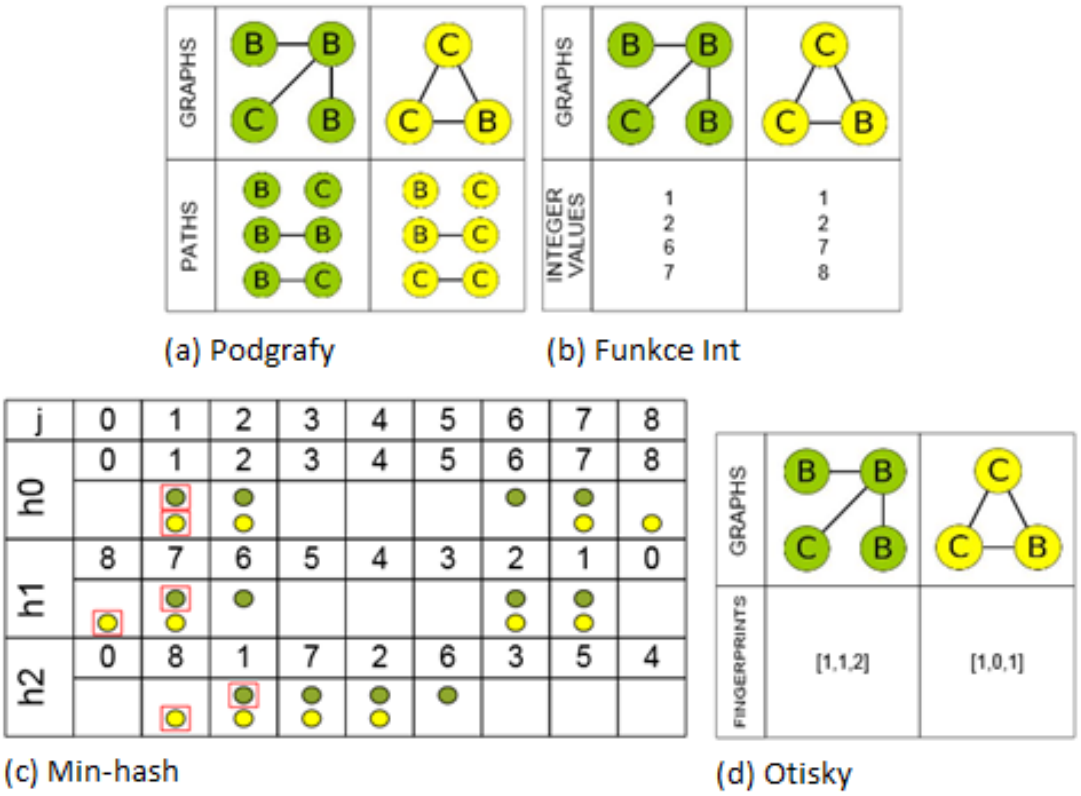
$$k_c^m(G,H) = \frac{\sum\limits_{p \in P_{(c)}} min(\phi_p(G), /phi_p(H))}{\sum\limits_{p \in P_{(c)}} max(\phi_p(G), /phi_p(H))}$$

Kde:

- G, H představují dva grafy
- $\phi_p(G)$ zastupuje funkci, která vrací počet výskytů cesty p v grafu G
- $P_{(c)}$ je množina všech možných cest velikosti maximálně c

Algoritmus Min Hash funguje v následujících fázích:

- Vytvoření podgrafů
- Výpočet funkce Int – může být provedeno několika způsoby. V ukázce i nové navržené metodě pro detekci plagiátu je použitá funkce Int, která mapuje cestu do int32 hodnoty
  $int([v_1, v_2, v_3, \dots v_n]) = ((a_1 L(v_1) + a_2 L(v_2) + a_3 L(v_3) + \dots + a_n L(v_n))(mod P)$
  Kde v představuje vrchol a náhodné číslo 0<a<P, P je „big prime“ – číslo, jehož velikost ovlivňuje počet kolizí a L představuje „label funkci“.
- Použití stanoveného počtu hashovacích funkcí na hodnoty bodu 2) $h_i(x) = (a_i x + b_i)(mod M)$ Kde a_i a b_i jsou náhodná čísla
  $0 < a_i < M$ $0 <= b_i < M$ M představuje „big prime“ $M >= P$
- Nalezení MinHash – min hash lze definovat pomocí následující rovnice:
  $h_i^{min}(set) = \{h_i(x) | x \in set \wedge h_i(x) \leq h_i(y), \forall y \in set\}$
- Vytvoření otisků – otisk v tomto případě představuje vektor min hash hodnot
- Porovnání pomocí MinMax



Obrázek 5: Ukázka vytvoření otisků pomocí Min Hash

```
Vstup:    Substruktury S(G) grafu G, Set hash funkci H
Vystup: Otisk

k = |H|;
    fingerprint [i] = ∞;
set = ∅ ;

foreach s ∈ S(G) do
    set = set ∪ int(s)

foreach i ∈ 1, 2, ... k do
    fingerprint [i] = h_i^min(set);
```

Výpis 2: Pseudokód Min Hash

Pravděpodobnost, že dvě množiny dat, v tomto případě dva grafy vygenerují stejné hodnoty min-hash vychází z Jaccardovy rovnice:

$Pr[h_i^{min}(A) = h_i^{min}(B)] = \frac{|A \cap B|}{|A \cup B|}$

Kde:

- A,B jsou množiny porovnávaných dat
- h_i představuje hashovací funkci, kterou můžeme vnímat jako náhodnou permutaci položek z $A \cup B$

Výpočet podobnosti vychází z faktu, že pravděpodobnost shodných minhash u množin A a B je roven pravděpodobnosti výběru prvku sdíleného A I B v $A \cup B$.
Vzorec pro výpočet podobnosti dvou grafů na základě min-hash hodnot má podobu:

$$sim(A, B) = \frac{\{1 \leq i \leq k \wedge h_i^{min}(A) = h_i^{min}(B)\}}{k}$$

Kde:

- A,B jsou množiny porovnávaných dat
- h_i představuje hashovací funkci, kterou můžeme vnímat jako náhodnou permutaci položek z $A \cup B$

Jinými slovy podobnost lze určit jako podíl shodných a rozdílných min-hash hodnot.

Otisky jsou pomocí min-hash generovány v čase O(nk) kde n představuje počet hash funkcí jenž byly použity.

4.5 Detekční nástroje

Tato kapitola obsahuje přehled nejpoužívanějších detekčních nástrojů spolu s popisem jejich vlastnosti a způsobem porovnávání souborů. Výkonnostní testy těchto nástrojů lze nalézt v literatuře [23, 24, 25].

4.5.1 JPlag

Detekční nástroj JPlag[4] byl vytvořen Guidem Malpohlem z univerzity Karlsruhe v rámci studentského projektu. Při porovnání využívá znalost syntaxe formátu kontrolovaného souboru díky čemuž je odolný vůči mnohým plagiátorským technikám. Od roku 2005 je JPlag přístupný také jako webová služba.

JPlag nejprve převede zvolené soubory do sekvencí tokenů, které reprezentují strukturu porovnávaného programu a následně tokeny porovnává dle optimalizovaného Greedy String Tiling algoritmu.

Systém porovnává a analyzuje programy vytvořené v programovacím jazyce Java, Scheme, C nebo C++. Výsledky porovnání prezentuje ve formě HTML dokumentů.

4.5.2 MOSS

Dalším velmi oblíbeným detekčním systémem je Measure Of Software Similarity (MOSS) z dílny Sanfordské Univerzity. Systém je dostupný ve formě webové služby, ke které lze přistupovat pomocí skriptu uveřejněného na internetové stránce [29].

Porovnání zdrojových kódů je prováděno rozdělením dokumentů do k-gramů, které představují řetězce znaků o pevné délce k. Každý k-gram je následně ohodnocen hashovací funkcí. K určení podobnosti souborů MOSS využívá výše popsaný algoritmus zvaný winnowing[5]. Ten vytváří podmnožiny těchto funkcí, které v podstatě představují otisk dokumentu, a porovnává je.

Systém analyzuje zdrojové kody programovacích jazyků C, C++, Java, C#, Python, Visual Basic, JavaScript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly a HCL2.

4.5.3 SIM

Detekční opensorce systém SIM pochází z roku 1989 a jeho autorem je Dick Grune. Porovnání podobnosti je prováděno v několika krocích. Nejprve dochází k tokenizaci zdrojových souborů pomocí lexikálního analyzéru. Tokeny identifikátorů jsou určovány dynamicky a ukládány do referenčních tabulek, které jsou déle použity pro určení míry podobnosti programů.

Systém porovnává zdrojové kódy programovacích jazyků C, Java, Pascal, Modula-2, Lisp a Miranda.

4.5.4 Plaggie

Detekční opensouce systém Plaggie[22, 30] je svým vzhledem i funkčností velice podobný systému JPlag. Vynikl v roce 2002 v technické univerzitě Helsinki a jeho tvůrcem je Ahtiainen et al. A (Ahtiainen et al. at Helsinki University of Technology)

Porovnání je prováděno tokenizací zdrojových kódu, vytvořením referenční tabulky a následným porovnáním tokenů pomocí Greedy String Tiling algoritmu.

Velkou nevýhodou systému je podpora pouze jednoho programovací jazyka a to Javy a možnost instalace pouze na operační systémy Unix a Linux.

5 Common intermediate language

Tato kapitola si klade za cíl seznámit čtenáře s problematikou Common intermediate language (CIL) a jejím využitím v systémech pro detekci plagiátu softwaru. Pro celkové pochopení problematiky generování a práci s CIL soubory dále doporučuji knihu „C# to IL„[31] jenž mi posloužila jako hodnotný zdroj informací a podkladů pro tuto diplomovou práci.

5.1 Definice CIL

V informačních technologiích představuje CIL dříve nazývaný také Microsoft Intermediate Language (MSIL) nejnižší člověkem čitelný programovací jazyk patřící do množiny.Net jazyků. Jedná se o objektově orientovaný programovací jazyky vyloženě zásobníkového typu. To v podstatě znamená, že data jsou ukládaná z registrů přímo do zásobníku a ne do paměti. Je procesorově a platformně nezávislý a může být používán v jakémkoli prostředí podporujícím Common Language Infrastructure (CLI). CIL a jeho instrukce jsou generovány kompilátorem nebo takzvaným IL assemblerem (ILASM). Takto vytvořený soubor lze znovu rozložit pomocí IL Disasembleru (ILDASM).

Obrázek 6: Celkový proces kompilace zdrojových kódu .NET

5.2 Motivace pro použití CIL

Použití CIL místo zdrojového kódu má u detekčních softwaru své opodstatnění. Kromě dříve již zmiňované možnosti otestovat soubory různých jazyků, CIL usnadňuje práci také díky normalizovanému textu bez komentářů a bílých znaků. Řetězce a číselné hodnoty, jež v CIL můžeme vidět, jsou mnohdy předzpracovány. Naopak CIL nezpracovává nepoužitelný kód tedy například kód za instrukcí return nebo za nesplnitelnou podmínkou a nepracuje ani s původními názvy proměnných.

Detekční nástroje používající CIL jsou založeny na sledování a porovnávání toku dat. Vychází z předpokladu, že jakkoli se syntaxe programů změní, tok dat musí ve většině případů zůstat stále stejný.

5.3 Struktura CIL

Každý CIL kód se skládá z direktivy následované atributy, kódem operace a labely.

Tokeny s prefixem „.“ například jmenné prostory, třídy, metody, konstruktory atd. jsou nazývány Direktivy CIL. Jejich účel je zřejmý – rozděluji kód do bloků dle třídy a metod, do kterých patří.

Za každou direktivou následují atributy CIL. Například: extends, implements, public,atd.

Kód operace také nazývaný CIL Opcodes představují tokeny, které se používají k sestavení implementační logiky. Jedná se o binární kódy (0x00), které však mají také uživatelský přívětivější podobu. V tabulce 3 lze vidět ukázku těch nejčastěji používaných a jim odpovídající instrukce a definice.

Tokeny ve tvaru IL_0000 neboli CIL Code labels představují pouze labely, které lze jednoduše nahradit nebo dokonce i vyloučit. Pokud je v IL kódu ponecháme, budou sloužit k upřesnění pozice například při procházení cyklem.

Kód operace	Instrukce	Definice
0x00	Nop	Neprovádí žádnou operaci.
0x01	Break	
0x01	ldarg.0	Uložení číselné hodnoty
0x72	ldstr	Uloží hodnotu string do zásobníku.
0x73	newobj	Vytvoření nového objektu
0x7A throw		

Tabulka 3: Ukázka kódu operací

5.4 Ukázky základních struktur

Pro lepší názornost je tato kapitola věnována praktickým ukázkám nejpoužívanějších CIL kódu a jejich vysvětlení.

5.4.1 Proměnné

CIL obsahuje instrukce, jak pro číselné, tak i textové proměnné. Přitom zde dochází k zjednodušování proměnných zobrazeném na obrázku **??** . CIL nebere v potaz názvy původních proměnných, ale přiřazuje jim vlastní názvy ve tvaru V_číslo. Datové typy jsou změněny z int na int32 a z long na int64.

Hodnota proměnné je vždy uložena do zásobníku. Pokud je nutné ji použít, musí být nejprve načtena ze zásobníku a teprve poté použita v rámci požadované operace. Ukázku lze vidět na výpisu 4 a 5.

```
IL_0001:  ldstr      "Hello␣World"
```

Výpis 3: Deklarace proměnné v CIL

```
static void add()
{
    int i = 10;
        int j = 20;
        int k = i + j;
}
```

Výpis 4: Ukázka kódu v C#

```
.method private hidebysig static void 'add'() cil managed
{
    // Code size 12 (0xc)
    .maxstack 2
    .locals init ([0] int32 value1,
             [1] int32 value2,
             [2] int32 value3)     // deklarace loaklnich promennych

    IL_0000: nop                   // zadna operace

    IL_0001: ldc.i4.s 10 //ulozeni int32 hodnoty do zasobniku

    IL_0003: stloc.0               //vytazeni prvni promenne ze zasobniku a jeji ulozeni do promenne s
        indexem 0

    IL_0004: ldc.i4.s 20 //ulozeni int32 hodnoty do zasobniku

    IL_0006: stloc.1 0          // vytazeni prvni promenne ze zasobniku a jeji ulozeni do promenne s
        indexem 1

    IL_0007: ldloc.0               // nacteni prvni lokalni promenne a ulozeni jeji hodnoty do zasobniku

    IL_0008: ldloc.1               // nacteni druhe lokalni promenne a ulozeni jeji hodnoty do zasobniku

    IL_0009: add                   //vytazeni prvnich dvou hodnot ze zasobniku a jejich secteni, vysledek
        je ulozen opet do zasobniku

    IL_000a: stloc.2               //vytazeni hodnoty ze zasobniku a ulozeni do promenne

    IL_000b: ret
} // end of method Program::'add'
```

Výpis 5: Ukázka kódu v CIL

5.4.2 Cykly a podmínky

Cykly i podmínky jsou v CIL realizovány pomocí labelu. Tato situace je ilustrována na obrázku 6.

```
static void IterationExample()
{
    int i = 0;
    while (i < 5)
    {
        i++;
    }
}
```

Výpis 6: Ukázka kódu v C#

```
.method private hidebysig static void IterationExample() cil managed
{
    // Code size       20 (0x14)
    .maxstack 2
    .locals init ([0] int32 i,
             [1] bool CS$4$0000)      // KROK1
    IL_0000: nop                      // KROK2
    IL_0001: ldc.i4.0                 // KROK3
    IL_0002: stloc.0                  // KROK4
    IL_0003: br.s       IL_000b       // KROK5
    IL_0005: nop                      // KROK12  // KROK24
    IL_0006: ldloc.0                  // KROK13
    IL_0007: ldc.i4.1                 // KROK14
    IL_0008: add                      // KROK15
    IL_0009: stloc.0                  // KROK16
    IL_000a: nop                      // KROK17
    IL_000b: ldloc.0                  // KROK6  // KROK18
    IL_000c: ldc.i4.5                 // KROK7  // KROK19
    IL_000d: clt                      // KROK8  // KROK20
    IL_000f: stloc.1                  // KROK9  // KROK21
    IL_0010: ldloc.1                  // KROK10 //KROK22
    IL_0011: brtrue.s   IL_0005 //KROK11 //KROK23
    IL_0013: ret
} // end of method Program::IterationExample
```

Výpis 7: Ukázka kódu v CIL

Způsob zpracování kódu uvedeného ve výpise 7 lze popsat v následujících krocích:

- Krok1 deklarace proměnných s indexem 0 a 1
- Krok2 žádná operace
- Krok3 Načtení int32 hodnoty do zásobníku na pozici 1
- Krok4 Dojde k vytažení hodnoty ze stacku a její uložení do lokální proměnné s indexem 0
- Krok5 přechod na řádek IL_0000b
- Krok6 Načtení lokální proměnné do zásobníku

- Krok7 Načtení lokální proměnné do zásobníku
- Krok8 Operace sečtení dvou proměnných získaných ze zásobníku v kroce 6 a 7
- Krok9 Vytažení hodnoty ze zásobníku a uložení do lokální proměnné
- Krok10 Načtení proměnné do zásobníku
- Krok11 Kontroluje, zda je hodnota větší než 0. Pokud ano, pokračuje bodem IL_0005 pokud ne pokračuje na řádek ret.
- Krok12 Žádná operace
- Krok13 Načtení lokální proměnné do zásobníku
- Krok14 Načtení lokální proměnné do zásobníku
- Krok15 Operace sečtení dvou proměnných získaných ze zásobníku v kroku 13 a 14
- Krok16 uložení hodnoty do lokální proměnné

5.5 Nástroje pro práci s CIL

5.5.1 IL Disasembler

K získání CIL kódu výborně poslouží nástroj sady Visual Studio zvaný IL Disassembler (ildasm.exe), který zpracovává přenosné spustitelné soubory (PE) obsahující kód intermediate language(IL), a vytváří z nich textový soubor obsahující CIL.

5.5.2 .Net Reflektor

Velmi užitečný nástroj, který umožnuje dekompilaci IL do vyšších programovacích jazyků (C#,Visual Basic,...).

6 Popis algoritmů vlastní aplikace

Před představením kompletního návrhu nového detekčního softwaru, budou v této kapitole popsány postupy a algoritmy použité v navrhovaném systému spolu s jejich principy a důvody použití. Naleznete zde však také problémy a komplikace, jenž zvolený přístup přináší a v neposlední řadě jejich řešení a dopad na efektivitu a časovou náročnost systému.

6.1 Základ aplikace

Pro vlastní aplikaci byla zvolena detekční metoda PDG a to z toho důvodu, že vykazuje nejlepší výsledky při vyhledání plagiátu ze všech metod uvedených v kapitole.... Avšak narozdíl od existujících nástrojů pro detekci plagiátu pomocí PDG bude navrhovaná metoda pro vytvoření grafu vycházet z Comon Intermediate Language (CIL). Díky tomu bude možné porovnávat aplikace napříč všemi programovacími jazyky .Net frameworku.

Postup detekční metody lze shrnout do tří bodů:

- předzpracování CIL dokumentů
- vytvoření PDG grafu
- vytvoření otisků
- procházení podgrafů a hledání podobnosti

6.1.1 Předzpracování CIL dokumentů

Jelikož neexistuje nástroj, jež by dokázal uspokojivě převést vygenerovaný CIL kód do požadované podoby PDG bude vytvořen prototyp parseru. Vzhledem k tomu, že se jedná o časově náročný úkol bude tento nástroj omezen na rozpoznání pouze nejpoužívanějších struktur. Pro použití v komplexnějších programech bude potřeba parser rozšířit o další specifika CIL kódu.

6.1.2 Vytvoření PDG grafu a otisků

Zásadní nevýhodou porovnání souboru pomocí PDG je mnohdy výpočetní časová náročnost této metody. Výpočetní čas radikálně zvyšuje poslední fáze porovnání grafů, jelikož zde narážíme na nedeterministicky polynomialni (NP) problém izomorfismu grafů. Vzhledem k tomu, že daný problém nelze vyřešit v polynomiálně omezeném čase je nutné výpočetní čas zkrátit použitím doplňující metody. V tomto případě se jako nejefektivnější jeví otisky neboli „fingerprints“. Základní jednotkou, z nichž budou otisky vytvářeny, je tok dat proměnné.

6.1.3 Zjištění míry podobnosti

Algoritmem pro určení míry podobnosti byl zvolen MinMax a to především pro jeho jednoduchost a efektivitu. V rámci tohoto algoritmu budou použity tři hashovací funkce, což je počet dostatečný pro relevantní výsledky.

7 Návrh aplikace pro detekci plagiátu

V této kapitole je popsán návrh a implementace nového detekčního nástroje, jenž má za úkol zjišťovat podobnost zdrojových kódu napsaných v .NET programovacích jazycích. Mým cílem bylo navrhnout a vytvořit aplikaci, která by nejen splňovala kritéria zadání diplomové práce, ale zároveň svým výkonem nijak nezaostávala za již existujícími nástroji. Navržená aplikace by měla sloužit k porovnání zdrojových kódu zkompilovaných do CIL. Nespornou výhodou a zároveň i důvodem použití CIL v této aplikaci je právě možnost porovnávat zdrojové kódy napsané v různých .NET programovacích jazycích.

Pro vytvoření CIL souborů lze použít nástroj sady Visual Studio zvaný IL Disassembler (ildasm.exe), který zpracovává přenosné spustitelné soubory (PE) obsahující kód intermediate language(IL) a vytváří z nich textový soubor obsahující CIL.

7.1 Analýza požadavků

Aplikace má za úkol srovnat dva vstupní programy napsané v .NET programovacích jazycích. Vstupním parametrem aplikace jsou tedy dvě cesty k zdrojovým souborům. První je považován aplikací za originál a druhý za podezřelý. Jako výstup aplikace se zobrazí oba porovnávané soubory ve formě PDG grafů, kde podobné podgrafy neboli plagiáty budou označeny stejnou barvou. Dále se v rámci výstupu zobrazí porovnávané CIL soubory, kde budou opět označeny podobné nebo dokonce shodné úseky.

Funkční požadavky:

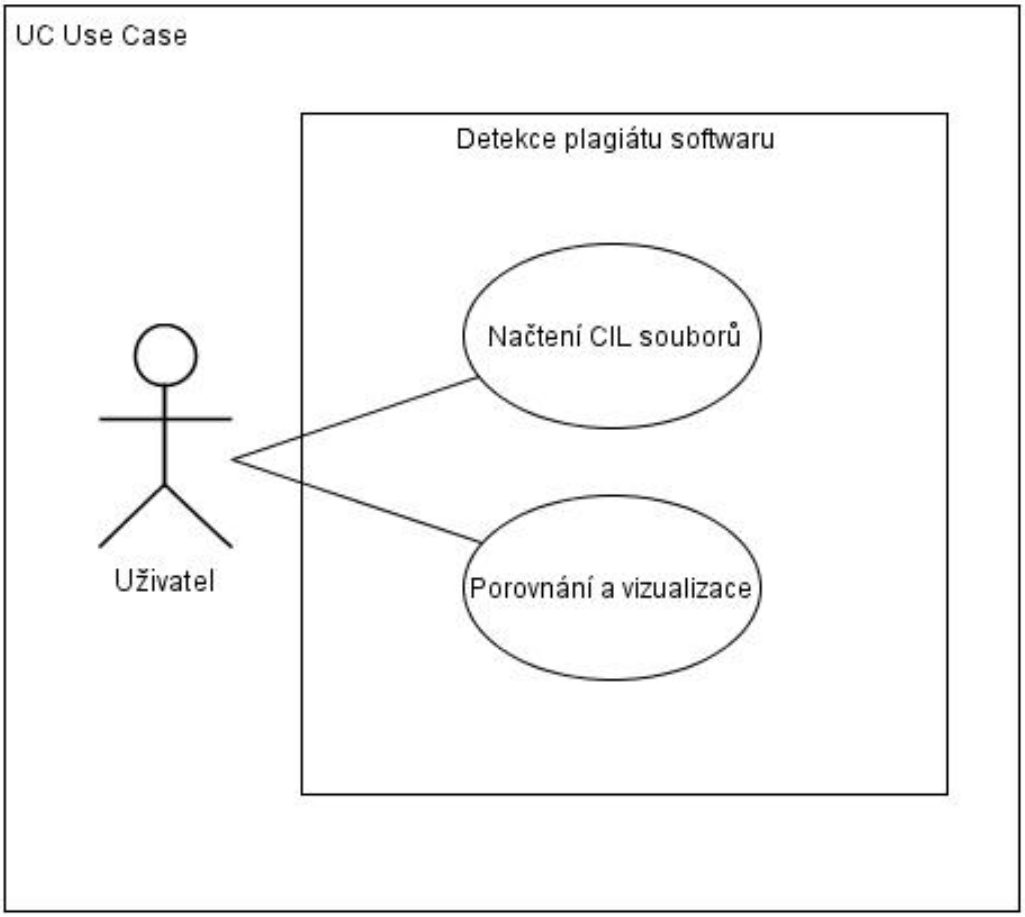
- aplikace bude načítat porovnávané CIL soubory z pevného disku
- aplikace bude informovat uživatele o stavu zpracování pomocí výpisů
- aplikace zobrazí graf podobnosti
- aplikace zobrazí podobnost CIL souborů

Ostatní požadavky:

- aplikace podporuje zdrojové kódy napsané v .NET programovacích jazycích
- přiměřená doba odezvy
- odolnost vůči technikám plagiátorů

7.2 Případy užití

Chování systému z hlediska uživatele je zachyceno v následujícím Use Case (US) diagramu neboli diagramu případů užití na obrázku 7.



Obrázek 7: Use Case Diagram

UC1 - Načtení CIL souborů
Uživatel zvolí CIL soubory, jež chce porovnat.

UC2 - Porovnání zdrojových kódu a vizualizace
Uživatel zvolí porovnání programů a zobrazení výsledných PDG Grafu a CIL souborů.

Případ užití: UC1 - Načtení porovnávaných souborů	
1	Uživatel stiskne záložku “Soubor”
2	Uživatel zvolí “Porovnat CIL”
3	Aplikace zobrazí formulář pro zadání cest k dokumentům
4	Uživatel zadá cestu ke zdrojovým souborům, které chce porovnat
5	Uživatel Stiskne tlačítko “Porovnat”
6	Aplikace načte porovnávané soubory
7	Aplikace zobrazí tlačítko “Porovnej” pro zahájení porovnání a vizualizaci výsledků
Rozšíření:	
3a	Nejedná se o CIL soubory
1	Aplikace upozorní uživatele na nesprávně zvolený soubor
2	Případ užití končí

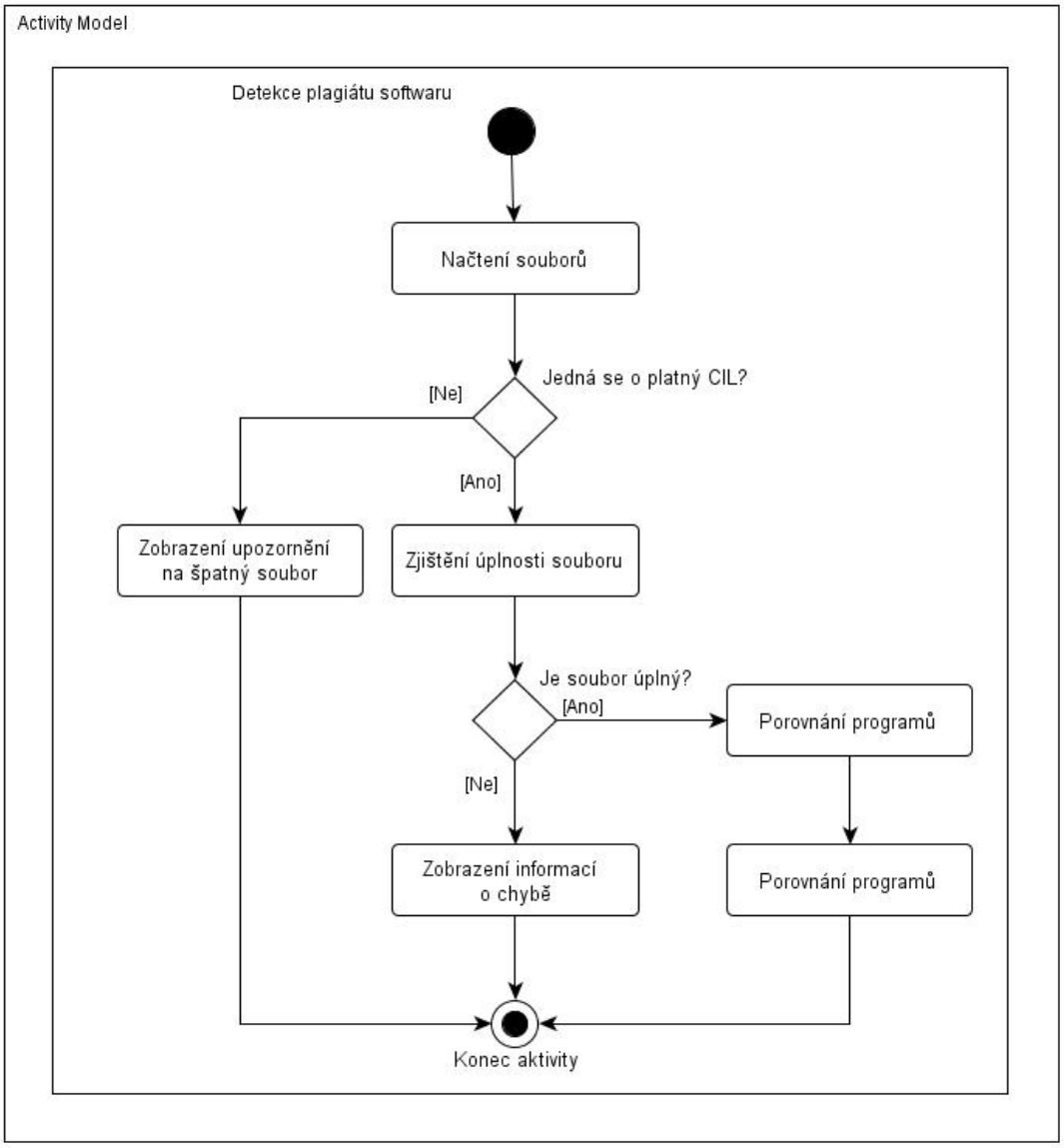
Tabulka 4: UC1 Načtení CIL souborů

Případ užití: UC1 - Porovnání zdrojových kódu a vizualizace	
1	Uživatel stiskne tlačítko “Porovnat”
2	Aplikace zpracuje načtené soubory
3	Aplikace informuje uživatele o úspěšnosti zpracování souborů
4	Aplikace vytvoří PDG grafy obou CIL souborů
5	Aplikace informuje uživatele o úspěšnosti vytvoření grafů
6	Aplikace porovná vytvořené PDG grafy
7	Aplikace vypočte podobnost mezi grafy
8	Aplikace zobrazí PDG grafy obou porovnávaných zdrojových kódu
9	Aplikace zobrazí CIL soubory obou porovnávaných zdrojových kódu
Rozšíření:	
2a	Načtený CIL soubor je chybný nebo nekompletní
1	Aplikace informuje uživatele o chybě
2	Případ užití končí

Tabulka 5: UC2 Porovnání zdrojových kódu a vizualizace

7.3 Diagram aktivit

Proces porovnávání zdrojových kódů zachycuje diagram aktivit na obrázku 8.



Obrázek 8: Diagram aktivit

7.4 Návrh aplikace

Cílem navržené aplikace je odhalit plagiát softwaru a to ať již se jedná o plagiát celého díla, nebo jen jeho části napříč různými .NET programovacími jazyky. Aplikace navíc musí být rezistentní vůči plagiátorským technikám používaným k zamaskování plagiátu.

První část procesu přestavuje předzpracování, které se skládá především z parsování CIL souborů. Procesem parsování získáme CIL instrukce pro jednotlivé třídy, metody a proměnné potřebné pro další fáze aplikace. Tento krok je o to jednodušší, že CIL soubor na rozdíl od samotného zdrojového kódu neobsahuje uživatelsky přívětivé avšak pro zpracování nebo automatizaci zcela zbytečné informace jako například uživatelské komentáře.

Následuje fáze zpracování instrukcí, kdy aplikace postupně prochází jednotlivé instrukce každé metody podobným způsobem jako CIL Assembler a zpracovává je dle jejich typu, funkce a vstupních proměnných. Výsledkem této fáze je kolekce tříd jejich metod a sady zpracovaných instrukcí obsahujících informace o typu instrukce, vstupních proměnných a toku dat. Z dat získaných v této fázi je následně vytvořen PDG graf.

PDG grafy jsou v třetí fázi rozděleny do podgrafů, kde jeden podgraf přestavuje tok dat pro konkrétní proměnnou. Pro každý graf tedy platí, že počet deklarovaných proměnných je stejný jako počet podgrafů. Každému podgrafu je vytvořen otisk, který obsahuje tři důležité informace: typ proměnné, Min-Hash podgrafu a Min-Hash toku dat pro vrcholy typu “control”.

Posledním krokem je proces samotné detekce, který je založený na nalezení MinMax hodnoty pro dané otisky. Dle výše této hodnoty je určeno, zda ověřované podgrafy jsou plagiáty nebo ne. Pokud budou podgrafy souboru, který je aplikací vnímán jako originál, shodné s podgrafy souboru označeného za podezřelý pak se jedná o plagiát s podobností 100%. V tomto kroku také probíhá zpětné mapování podobnosti na ověřované CIL soubory, díky čemuž můžeme v uživatelském rozhraní přesně vidět, které části byly duplikovány.

Obecně lze proces aplikace popsat v několika krocích:

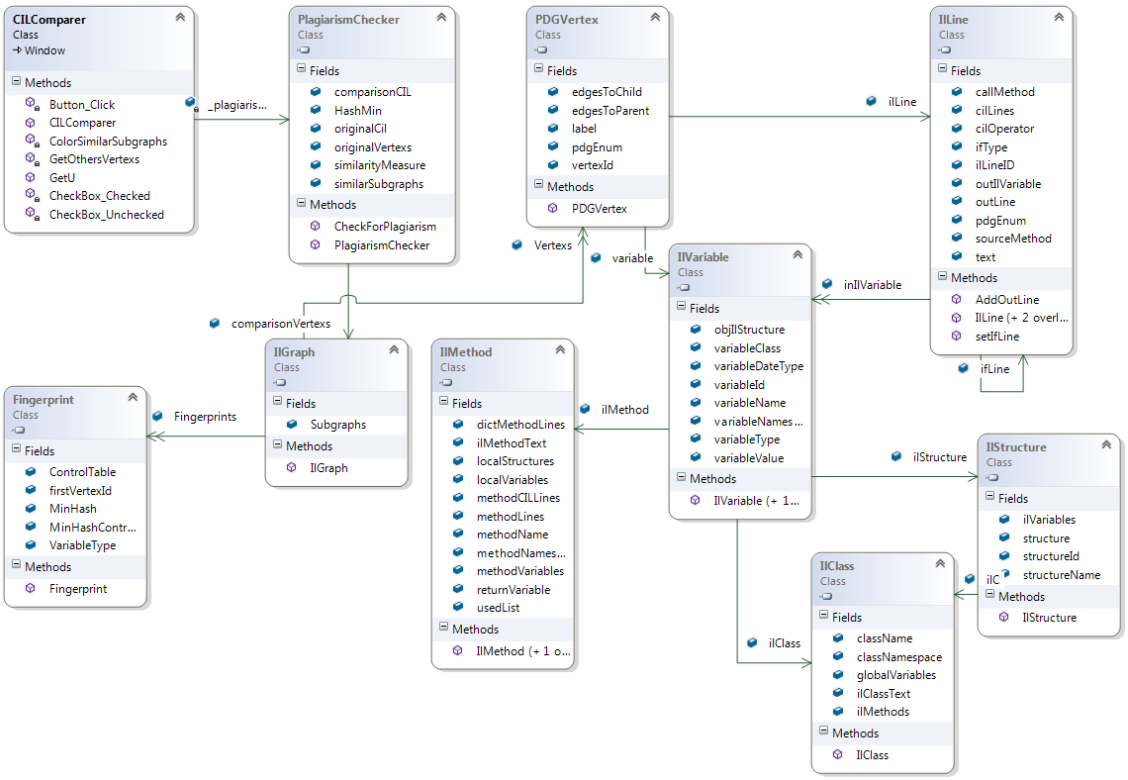
- Předzpracování souborů - rozdělení CIL souboru na třídy, metody, parametry a instrukce
- Zpracování instrukcí - zpracování CIL instrukcí do podoby obsahující typ instrukce, vstupní proměnné, tok dat
- Vytvoření PDG - vytvoření PDG ze zpracovaných instrukcí
- Vytvoření podgrafů a otisků - vytvoření podgrafů pro každou proměnnou a přiřazení otisků dle typu proměnné a MinHash funkcí
- Porovnání otisků - porovnání otisků dle funkce MinMax

Obrázek 9: Struktura navržené aplikace

- Vizualizace výsledků - zobrazení podobnosti otisků, porovnání PDG grafů se zvýrazněnými podobnými podgrafy a zobrazení porovnání CIL souboru se zvýrazněnými podobnými úseky

7.5 Třídní diagram aplikace

Na obrázku 10 lze vidět kompletní třídní diagram navržené aplikace.



Obrázek 10: Třídní diagram aplikace

Třída PlagiarismChecker představuje ústřední třídu pro porovnání CIL souboru, jelikož zprostředkovává komunikaci s ostatními třídami v projektu. Z této třídy následně prezenční vrstva získává informace o míře podobnosti celého grafu i jednotlivých podgrafů.

Třída CILParser obsahuje metody pro předzpracování CIL souborů a následné zpracování instrukcí. Tato třída je přizpůsobená primárně pro CIL, jelikož se jedná o soubory s velice specifickou strukturou.

PDGCreater obsahuje veškeré potřebné operace pro vytvoření PDG grafu. Při výpočtech vychází z výsledku resp. zpracovaných instrukcí třídy CILParser,

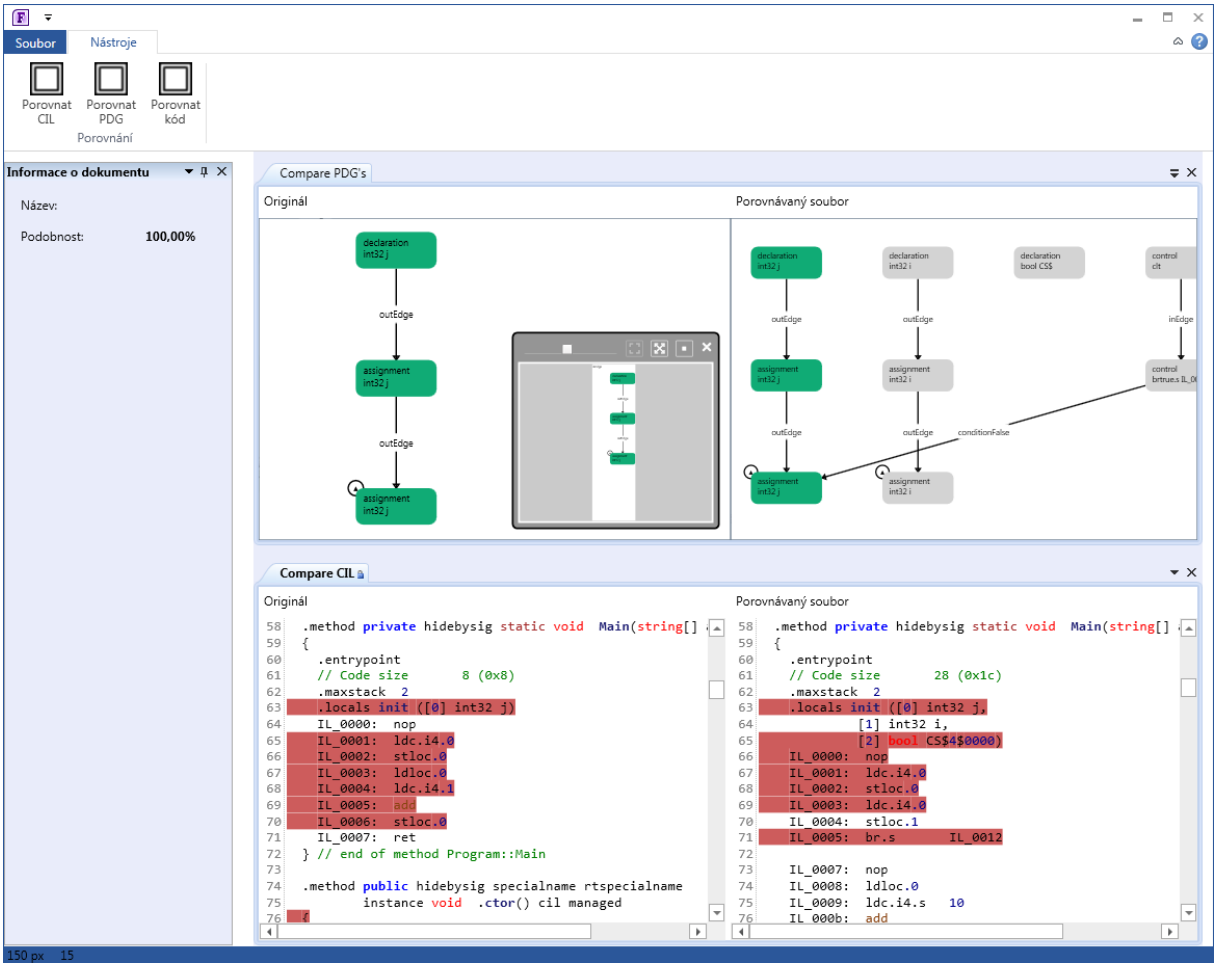
Následuje proces porovnání PDG grafů, který se odehrává ve třídě CILComparator. Nejprve jsou vytvořeny podgrafy pro oba porovnávané soubory, následně jsou podgrafům přiřazeny otisky a právě ty se v další fázi porovnávají. Třída obsahuje algoritmus Hash-Min pro vytvoření otisků a MinMax pro zjištění podobnosti podgrafů.

Celý projekt využívá datovou vrstvu, která je, stejně jako komponenta PDGGraph, z důvodu znuvupoužitelnosti umístěna do samostatné knihovny PlaGSW.Model.

Hlavní třídu prezenční vrstvy komunikující s uživatelem v tomto případě přestavuje MainWindow. Obsahuje a volá metody potřebné pro porovnání CIL souborů a zobrazení podobnosti pomocí PDG včetně barevného označení podobných podgrafů. Vzhledem k tomu, že aplikace pracuje se soubory CIL, nemusí uživatel specifikovat v jakém programovacím jazyce byl zdrojový kód napsán. Pro porovnání stačí pouze nahrát dva CIL soubory. Třída MainWindow využívá dále zmiňované knihovny AvalonDock a Fluent Ribbon Control Suite. Součástí prezenční vrstvy je také uživatelský ovládací prvek PDGGraph, který je, jak již bylo řečeno, z důvodu zpřehlednění a znuvupoužitelnosti umístěn do samostatné knihovny.

7.6 Grafické rozhraní

Pro účely zobrazení výsledků navrhované metody postačí jednoduché grafické rozhraní navržené ve Windows Presentation Foundation (WPF) frameworku pro komplexní tvorbu formulářových aplikací, který je součástí Visual Studia od verze 3.0. Pro vytvoření dokovacích oken bude použita WPF knihovna AvalonDock, která umožňuje vytvářet jednoduchý dokovací systém. A v neposlední řadě pro zpříjemnění celkového vzhledu aplikace bude použita knihovna Fluent Ribbon Control Suite, která umožní vytvořit uživatelské rozhraní pro WFP velmi podobné standardnímu rozhraní aplikací Microsoft Office.



Obrázek 11: Ukázka grafického prostředí aplikace

8 Implementace

Tato kapitola pojednává o aplikačních algoritmech navrženého systému pro detekci plagiátu. Jednotlivé podkapitoly popisují algoritmy použité v aplikaci, jejich strukturu a účel.

8.1 Inicializace aplikace

Při prvotní inicializaci jsou načteny veškeré komponenty prezentační vrstvy. Teprve poté, co uživatel zadá (do speciálně pro tento účel vytvořeného okna CompareCILWindow) cesty k porovnávaným souborům, spustí se proces porovnání CIL. Vzhledem k tomu, že je aplikace zaměřena na práci s CIL soubory jejichž struktura je pro všechny .NET programovací jazyky společná, není potřeba specifikovat, v jakém konkrétním jazyce je daný zdrojový soubor napsán.

Pro rozšíření aplikace o porovnání PDG grafů z jiných zdrojů než CIL by bylo potřeba vytvořit nový parser pro daný jazyk nebo zdroj. Na takto zpracovaný graf by pak bylo možné použít metody třídy CILComparator pro porovnání podgrafů a zjištění podobnosti.

8.2 Porovnání souborů

Po načtení programu a zvolení porovnávaných souborů je spuštěn proces porovnávaní souborů. Nejprve je tedy vytvořena instance třídy PlagiarismChecker a následně spuštěná její metoda CheckForPlagiarism.

```
public double CheckForPlagiarism(string path, string comparisonPaths)
        {
            var cilParser = new CILParser();

             originalCil = cilParser.GetCIL(path);
            var originalMainMethods = new List<IlMethod>();
            var originalmainMethod = cilParser.PrepocessIL(originalCil, out originalMainMethods);
            originalVertexs = PDGCreater.CreatePDGGraph(mainMethod);

            cilParser = new CILParser();
            comparisonCIL = cilParser.GetCIL(comparisonPaths);
            var comparisonMainMethods = new List<IlMethod>();
            var comparisonMainMethod = cilParser.PrepocessIL(comparisonCIL, out
                comparisonMainMethods);
            comparisonVertexs = PDGCreater.CreatePDGGraph(comparisonMainMethod);

            var similarity = CILComparator.CompareGraphs(originalVertexs, comparisonVertexs);
            HashMin = (similarity[0]*100).ToString("0.00");

            similarSubgraphs = CILComparator.similarSubgraphs;

            return similarity;
        }
```

Výpis 8: Metoda vracející podobnost grafů resp. Cil souborů

Ve výpisu 8 je uvedena řídící metoda pro získání podobnosti grafů resp. CIL souborů. Metoda GetCil třídy CILParser nejprve načte a vrátí CIL soubor v jeho surové podobě. Následně dojde k předzpracování CIL souboru a zpracování jeho instrukcí díky metodě PreprocessIL třídy CILParser, která v tomto případě vrací Objekt IlMethod obsahující hlavní metodu CIL souboru. Následně je objekt IlMethod zpracován metodou CreatePDGGraph, která ze získaných dat vytvoří PDG graf. Celý tento proces se opakuje také pro druhý soubor označený jako podezřelý. Poté co jsou získány PDG grafy obou souborů je spuštěna metoda CompareGraphs třídy CILComparator která jak již lze z názvu vytušit, porovnává PDG grafy a to tak, že vytváří pomocí funkce MinHash otisky pro jednotlivé grafy a pomocí funce MinMax a vrací hodnotu jejich podobnosti.

8.3 Parsování CIL souborů

V rámci parsování souborů dochází, jak již bylo naznačeno výše, k předzpracování CIL souboru a následném zpracování jeho jednotlivých instrukcí.

V rámci předzpracování je použito několik regulérních výrazů k získání tříd, metod a parametrů používaných v rámci testovaného projektu.

```
private const string regexPatternCilClass = @"\.class ((.|\ n)+?(?=end␣of␣class))";
private const string regexPatternCilNamespaceAndName = @"\.class((.|\n)+?(?=␣extends))";
private const string regexPatternGlobalVariable =@"\.field.*?(\n |\. method|\. field |\. class)";
private const string regexPatternLocalVariable = @"\.locals␣init␣ \(([^) ]*) \) ";
private const string regexPatternCilMethod = @"\.method((.|\n)+?(?=end␣of␣method))";
private const string regexPatternMethodHeader = @"\.method(.|\n)*{";
private const string regexPatternMethodBody = @"(?<=\{)(.|\n)*(?=\})";  //{(.|\ n)*?}
private const string regexPatternMethodName = @"[^\\␣]*\(.*\)";
private const string regexPatternMethodRow = @"IL_[0-9]*[a-zA-Z]*:";
private const string regexPatternMethodLines = @"IL(.|\n)*(?=\})";
```

Výpis 9: Regulární výrazy použité při parsování CIL souboru

Ve výpisu 9 jsou zobrazeny základní regulární výrazy použity k získání struktury CIL dokumentu jeho tříd, metod a parametrů. Regulární výraz regexPatternCilClass získá z CIL souboru části představující jednotlivé třídy. Regulární výraz používající regexPatternCilNamespaceAndName slouží k získání názvu jmenného prostoru a třídy. Zatímco regexPatternGlobalVariable je výraz určený k vyčtení globálních proměnných třídy, regexPatternLocalVariable slouží pro zjištění lokálních proměnných. Dále následuje několik výrazů pro získání informací o metodách. Například regexPatternMethodHeader načte hlavičku metody, která v CIL souboru mimo jiné obsahuje název metody, její jmenný prostor a proměnné. Jméno metody z této hlavičky můžeme získat použitím regexPatternMethodName. Zpracování těla metody nám umožní regexPatternMethodBody. Nakonec k získání instrukcí dané třídy postačí regexPatternMethodRow a regexPatternMethodLines.

Typ	Popis
declaration	Deklarace proměnné nebo struktury
assignment	Načtení proměnné
call	Volání metody
control	If, switch, while, do-while, for

Tabulka 6: Typy instrukcí

Typ	Popis
out	Představuje hlavní tok dat
in	Představuje vstupní proměnnou
true	Představuje pokračování podmínky nebo cyklu
false	Představuje pokračování podmínky nebo cyklu

Tabulka 7: Typy toku dat

Takto připravená data dále používá metoda pro zpracování instrukcí preprocessCILMethodBody, která prochází jednotlivé instrukce a dle typu a vstupních parametrů vytváří instance třídy IlLine, které představují jakýsi ekvivalent pro jeden, nebo více řádků instrukcí CIL dokumentu.

Jinými slovy, pokud instrukce pouze předává hodnotu do zásobníku, vytvoří se pouze instance třídy IlVariable a vloží se do zásobníku. Všechny ostatní instrukce, které ovlivňují tok lokálních nebo globálních proměnných v programu jsou převedeny dle jejich vlastností na jeden z typů uvedených v tabulce 7. Ke každé takovéto instrukci je vytvořena instance IlLine a jsou jí přiřazeny parametry dle toho jaké proměnné byly při zpracování instrukce použity.

```
IL_0006:  ldloc.0
IL_0007:  ldc.i4.1
IL_0008:  add
IL_0009:  stloc.0
```

Výpis 10: Ukázka CIL kódu

Výše popsaný případ je ilustrován na výpisu X. Instrukce IL_0006 zapisuje hodnotu lokální proměnné s indexem 0 do zásobníku a instrukce IL_0007 do zásobníku přidá číselnou hodnotu. IL_0008 obsahuje operátor add což znamená, že sčítá dvě poslední hodnoty uložené v zásobníku. Při zpracování instrukce IL_0008, jelikož se jedná pouze o součet, který prozatím nijak neovlivňuje tok lokálních nebo globálních proměnných, bude do zásobníku na poslední pozici uložena dvojice proměnných. Až instrukce IL_0009 vytvoří instanci IlLine jelikož se jedná o načtení hodnoty ze zásobníku do lokální proměnné - tudíž změnu toku lokálních dat.

8.4 Vytvoření PDG grafu

Vytvoření samotného PDG grafu po předešlých krocích již není nic složitého. Objekty IlLine vzniklé při parsování CIL dokumentu v podstatě představují vrcholy PDG grafu a IlLine.inIlVariable a IlLine. outIlVariable představují tok dat neboli hrany grafu.

8.5 Porovnání PDG grafů

Před samotným porovnáním je potřeba rozdělit graf na jednotlivé podgrafy. Pro potřeby navrhované aplikace jsem zvolila jednoduchý postup, který ale nejlépe reprezentuje tok dat v grafu. Jeden podgraf v tomto případě představuje tok dat jedné proměnné od její deklarace až po případný zánik.

Takto připraveným podgrafům následně vytvoří metoda CompareGraphs třídy CILComparator otisky. Každý z těchto otisků je tvořen ze třech základních informací: typ proměnné, MinHash podgrafu a MinHash toku dat pro vrcholy typu "control". Typ proměnné je důležitý zejména proto, že shodné grafy s rozdílnými podtypy jsou pro aplikace méně podobné než shodné grafy s proměnnými stejného typu. MinHash podgrafu v tomto případě představuje tok dat pro jednu proměnnou a je tedy také pro nalezení plagiátu nezbytný. Poslední informace obsažená v otisku MinHash toku dat pro vrcholy typu "control" je důležitá ze vzhledu na propojení jednotlivých toků dat v grafu.

9 Testování detekčních systémů

Teto kapitola je věnována testování nově vytvořeného detekčního systému. Na základě zjištěných poznatků lze vyhodnotit jeho použitelnost, eleftivitu a přesnost zvolených algoritmů. Vlastní aplikace bude srovnána s již existujícím nástrojem MOSS. U ostatních detekčních systému bude uvedena pouze odhadovaná úspěšnost jelikož nepracují s žádným z programovacích jazyků .NET.

9.1 Testovací data

Testovací data pochází z několika zdrojů. První sadu dat tvoří uměle vytvořené programy pro ověření všech základních plagiátorských technik. Druhá sada obsahuje rozsáhlejší projekty obsahující různé kombinace plagiátorských technik. Pro srovnání výsledků se současnými plagiátorskými nástroji byl zvolen MOSS jelikož dokáže zpracovat zdrojové kódy napsané v programovacím jazyce C#.

9.2 Testovací metodika

V rámci testování nástroje MOSS je vždy použita dvojice projektů, kde jeden představuje originál a druhý plagiát. Při testování vlastní aplikace jsou pak použity CIL kódy těchto projektů. V rámci testů jsou především ověřovány následující plagiátorské techniky:

- přepsání cyklů
- změna podmínek
- nahrazení těla metody za volání
- přidání kódu
- doplnění komentářů
- další syntaktické změny

9.3 Výsledky testování

Dle očekávání vykazuje nově navržený nástroj lepší výsledky zejména u syntaktických změn. Plagiát nezamaskují ani komentáře a nedosažitelný kód jelikož nejsou v CIL obsaženy. Výsledky nástroje MOSS jsou podstatně horší. V následující tabulce lze vidět v kolika procentech je druhý soubor považován za plagiát.

Popis	Vlastní aplikace	MOSS
přidání kódu navíc	100%	46%
záměna cyklu for za while	100%	15%
záměna podmínky if za switch	83,33%	11%
záměna podmínky if za ternary if	100%	11%
záměna metody v těle za volání if	100%	17%
záměna lokální proměnné za globální	100%	90%
přidání komentářů	100%	30%

Tabulka 8: Výsledky testování

Navržená aplikace funguje spolehlivě až na případ záměny podmínky if za switch. Jelikož nástroj MOSS nedosáhl uspokojivých výsledků na ukázkových příkladech lze očekávat, že ani v komplexnějších programech nebude příliš úspěšný. Srovnatelných výsledků by dokázal dosáhnout nástroj založený na porovnání PDG zdrojového kódu nebo AST stromů avšak žádný ze současně dostupných neplacených a fungujících nástrojů nezpracovává C# kód.

10 Závěr

Cílem práce bylo zmapovat současné nástroje a techniky detekce plagiátu a v neposlední řadě navrhnout a implementovat nástroj, jež by zdokonalil proces odhalování plagiátu mezi programy. Na základě teoretického výzkumu a analýz plagiátorských technik byl tedy navržen nový přístup, jenž by měl být odolný vůči základním i komplexnějším úpravám zdrojových kódu. Narozdíl od většiny současných nástrojů využívá CIL a otisky, což aplikaci umožnuje porovnávat zdrojové kódy libovolných programovacích jazyků .NET. Jelikož v době psaní této práce nebyl k dispozici žádný veřejně dostupný systém pro převod CIL do požadovaného tvaru PDG byl vytvořen také prototyp parseru, jež dokáže zpracovat základní struktury CIL kódu.

Výsledky závěrečného testování prokázaly odolnost nové aplikace vůči základním plagiátorským technikám a její efektivita je dokonce vyšší než u druhého testovaného nástroje MOSS. Námětem na rozšíření nebo doplnění této diplomové práce by tedy mohlo být zdokonalení převodu CIL na PDG grafy a aplikace vytvořeného nástroje na větší objem dat. Pro zvýšení využitelnosti aplikace by bylo možné také přidat vizualizaci původního zdrojového kódu a jeho porovnání.

11 Reference

[1] *Definice plagiátu*[online], 2013, [cit. 2013-03-25]. Dostupné z: <http://www.infogram.cz/findInSection.do?sectionId=1115&categoryId=1161>

[2] Němečková, L., *Plagiátorství*, Ústřední knihovna ČVUT, 2009, [cit. 2013-02-21]. Dostupné z: <http://knihovna.cvut.cz/studium/jak-psat-vskp/doporuceni/plagiatorstvi/>

[3] Clough, P., *Plagiarism in natural and programming languages: an overview of current tools and technologies*, Department of Computer Science, University of Sheffield, England, 2000.

[4] Prechelt, L., Malpohl, G., Philippsen, M. *JPlag: Finding plagiarisms among a set of programs*, Department of Informatics, University of Karlsruhe, Germany, 2000.

[5] Schleimer, S., Wilkerson, D. S., Aiken, A., *Winnowing: Local algorithms for document fingerprinting*, Computer Science Divison, UC Berkeley, California, 2003.

[6] Arwin, Ch., Tahaghoghi, S. M. M., *Plagiarism detection across programming languages*, School of Computer Science and Information Technology RMIT University, Australia, 2006.

[7] Jones, E. L., *Metrics based plagiarism monitoring*, Department of Computer and Information Sciences Florida A&M University Tallahassee, Florida, 2001.

[8] Halstead, M. H., *Elements of Software Science*, Elsevier Science Inc. New York, ISBN 0444002057, 1977.

[9] Shen, V. Y., Conte, S. D., Dunsmore, H. E., *Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support*, Department of computer science, Purdue University, 1981.

[10] Ghani, A. A. A., Hunter, R., *An Attribute Grammar Approach to specifying Halstead's Metrics*, Malaysian Journal of Computer Science, roč. 9, č. 1, s. 56-67, 1996.

[11] Salt, N. F., Conte, S. D., Dunsmore, H. E., Shen, V. Y., *Defining Software Science Counting Strategies*, ACM SIGPLAN Notices, roč. 17, č. 3, s. 58-67, 1982.

[12] Conte, S. D., Dunsmore, H. E., Shen, V. Y., *Software Engineering Metrics and Models*, Benjamin-Cummings Publishing Co., USA, ISBN 0805321624, 1986.

[13] Guoqiang, Z., *An experimental study on constructing sense relations in vocabulary teaching*, JCIT, čis. 4, s. 116-121, 2009.

[14] Pandey, K. L., Agarwal, S., Misra, S., Prasad, R., *Plagiarism Detection in Software Using Efficient String Matching*, Proceedings of the 12th international conference on Computational Science and Its Applications, roč. IV, s. 147-156, ISBN 9783642311277, 2012.

[15] Thathoo, R., Virmani, A., Lakshmi, S. S., Balakrishnan, N., Sekar, K., *TVSBS: A fast exact pattern matching algorithm for biological sequences*, Current Science Association, roč. 1, s. 47–53, 2006

[16] Berry, T., Ravindran, S., *A fast string matching algorithm and experimental results*, Software - practice and experience, roč. 25(7), s. 727-765, 1995 Proceedings of the Prague Stringology Club Workshop 1999, Collaborative Re-port DC-99-05, s. 16-26, 2001.

[17] Sheik, S. S., Aggarwal, S. K., Poddar, A., Balakrishnan, N., Sekar, K., *A FAST pattern matching algorithm*, Journal of Chemical Information and Computer Sciences, čis. 44(4), s. 1251–1256, 2004

[18] Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L., *Clone detection using abstract syntax trees*, Proceedings IEEE ICSM 1998, s.368-377, 1998.

[19] Ligaarden OS, *Detection of plagiarism in computer programming using abstract syntax trees*, Master Thesis, University of Oslo, Norway, 2007.

[20] Resmi N. G., Soman, K. P., *Abstract Syntax Trees with Latent Semantic Indexing for Source Code Plagiarism Detection*, International Journal of Advanced Research in Computer Science, čis. 3, s. 546, 2012.

[21] Liu, C., Chen, C., Han, J., *GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis*, Department of Computer Science, University of Ilinois-UC, Ilinois, 2006.

[22] Ahtiainen, A., Surakka, S., Rahikainen, M., *Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises*, Proceedings of the 6th Baltic Sea conference on Computing education research, s. 141–142, USA, 2006.

[23] Luke, D., Divya, P. S., Johnson, S. L., Sreeprabha, S., Varghese, E. B., *Software Plagiarism Detection Techniques:A Comparative Study*, International Journal of Computer Science and Information Technologies, čis. 5 (4), 2014.

[24] Juričić, V., Jurić, T., Tkalec, M., *Performance Evaluation of Plagiarism Detection Method Based on the Intermediate Language*, Department of Information Sciences, Faculty of Humanities and Social Sciences, University of Zagreb.

[25] Kleiman, A. B., Kowaltowski, T., *Qualitative Analysis and Comparison of Plagiarism-Detection Systems in Student Programs*, Instituto de Computa ao, Universidade Estadual de Campinas, Brazil, 2009.

[26] Wise, M. J., *String Similarity via Greedy String Tiling and running Karp-Rabin Matching* [online], 1993, [cit. 2013-04-06]. Dostupné z: <`http://opstools.googlecode.com/svn-history/r31/trunk/ndd/paper/`>

[27] Nilsson, E. A., *Abstract Syntax Tree Analysis for Plagiarism Detection*, Department of Computer and Information Science, Linköpings universitet, Sweden, 2012.

[28] Teixeira, C. H. C., Silva, A., Meira, W., *Min-Hash Fingerprints for Graph Kernels: A Trade-off among Accuracy, Efficiency, and Compression*, Computer Science Department, Universidade Federal de Minas Gerais, Brazil, 2012.

[29] *Internetová prezentace detekčního nástroje MOSS*[online]. Dostupné z: <http://www.cestadomu.cz/cz/nejcastejsi-dotazy.html>

[30] *Internetová prezentace detekčního nástroje Plaggie*[online]. Dostupné z: <http://www.cs.hut.fi/Software/Plaggie/>

[31] Mukhi, S., Safar, A., *C# to IL*[online]. Dostupné z: <http://vijaymukhi.com/>